MirChecker: Detecting Bugs in Rust Programs via Static Analysis

Zhuohua Li
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Jincheng Wang
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Mingshen Sun
Baidu Security

John C.S. Lui
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

ABSTRACT

Safe system programming is often a crucial requirement due to its critical role in system software engineering. Conventional low-level programming languages such as C and assembly are efficient, but their inherent unsafe nature makes it undesirable for security-critical scenarios. Recently, Rust has become a promising alternative for safe system-level programming. While giving programmers fine-grained hardware control, its strong type system enforces many security properties including memory safety. However, Rust's security guarantee is not a silver bullet. Runtime crashes and memory-safety errors still harass Rust developers, causing damaging exploitable vulnerabilities, as reported by numerous studies [29, 42, 47, 53, 54].

In this paper, we present and evaluate MirChecker, a fully automated bug detection framework for Rust programs by performing static analysis on Rust's Mid-level Intermediate Representation (MIR). Based on the observation of existing bugs found in Rust codebases, our approach keeps track of both numerical and symbolic information, detects potential runtime crashes and memory-safety errors by using constraint solving techniques, and outputs informative diagnostics to users. We evaluate MirChecker on both buggy code snippets extracted from existing Common Vulnerabilities and Exposures (CVE) and real-world Rust codebases. Our experiments show that MirChecker can detect all the issues in our code snippets, and is capable of performing bug finding in real-world scenarios, where it detected a total of 33 previously unknown bugs including 16 memory-safety issues from 12 Rust packages (crates) with an acceptable false-positive rate.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**;
• **Security and privacy** → *Software and application security*.

KEYWORDS

static analysis, Rust, abstract interpretation

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484541

ACM Reference Format:
Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3484541

1 INTRODUCTION

With the increase of software complexity, creating and maintaining high-quality software has become notoriously difficult for developers. Software failures in security-critical scenarios may cause devastating consequences. As an emerging programming language, Rust provides a scalable and systematic way of dealing with software errors. It integrates practical experience and research outcomes of programming languages in the past several decades, enabling programmers to build efficient programs safely. Its rigorous type system and the unique *ownership* system can rule out many memory safety bugs at compile time. Different from other safe programming languages, Rust is designed to be capable of low-level hardware manipulations, and therefore it is a promising language for system programming. Many companies and researchers are rewriting their low-level components and embedded systems in Rust, some of which have been a great success, such as the web browser engine Servo [1], the operating system Redox OS [17] and Tock OS [39]. Android Open Source Project also supports Rust for developing low-level OS components [51], and the adoption of Rust into the Linux kernel is currently in progress [25, 40].

Although Rust makes a significant step towards secure system programming, it is not a panacea. Empirical studies [29, 42, 47, 53, 54] have shown that Rust still has a considerable amount of security issues. First, to provide more flexibility, Rust has an `unsafe` keyword as an escape hatch, which enables extra powers for developers. While unsafe code is necessary especially for low-level operations such as dereferencing raw pointers, it may break the security guarantees and become the source of vulnerabilities. Existing studies have shown that unsafe code is now the most common cause of memory safety issues [53]. Second, even though one confines to pure safe Rust, sometimes memory safety is guaranteed by stopping program execution. For example, statically checking array bounds is in general not feasible. Therefore, the Rust compiler generates appropriate assertion statements to check these kinds of security conditions at runtime. If an assertion statement fails, a runtime panic is triggered. While aborting at runtime successfully prevents memory corruption, however, for security-critical applications, "no-panic policy" is usually enforced and therefore runtime crashes are considered unacceptable.

To tackle the above problems, we propose using static analysis to detect potential memory-safety issues and runtime panics before the Rust programs are deployed. Our approach can be regarded as a complement of Rust's type system, enforcing more rigorous security checks on Rust code. We argue that making a static analyzer dedicated for Rust by leveraging Rust's type system has the following advantages: (1) Precision: Rust is statically and strongly typed, so the type system can provide more information to make the analysis more accessible and more precise. A dedicated analyzer can also take advantage of the special patterns of bugs that are unique to Rust programs (§ 3.1). (2) User-friendliness: The Rust compiler explicitly inserts assertions to check safety conditions dynamically in order to prevent undefined behaviors. These assertions can be used by the static analyzer as conditions that should be checked (§ 4.2.3), thus no manual annotations are needed. (3) Efficiency: The *ownership* system statically determines the lifetime of each variable, so the analyzer can safely clean up the storage for variables that have gone out of their scopes. This dead variable cleaning mechanism reduces memory consumption and speeds up the analysis (§ 8.3).

In this work, we present MirChecker, a fully automated bug detection tool designed for Rust programs. It is based on the theory of *Abstract Interpretation* [18–20], which provides the foundation of dataflow analysis by formalizing the relationship between analysis and semantics. The core design of MirChecker follows the monotone framework [46] that approximates each property of interest as a complete lattice. *Transfer functions* are defined for each statement, specifying how the properties are manipulated and transferred by each statement. We propose to use a dedicated *abstract domain* that gathers both numerical and symbolic values, where the former is used for integer bounds analysis and the latter is mainly used as the memory model. Then a fixed-point algorithm is executed to propagate the properties through the *control flow graph* (CFG). When the fixed-point algorithm terminates, it generates the invariants for the whole program. Finally, we implement several bug detectors that leverage the analysis result to perform bug detection. MirChecker is implemented as an additional analysis pass of the official Rust compiler, and is integrated with the official package manager `Cargo` to provide a similar user interface with existing tools used by Rust developers. The analysis is done on top of Rust's Mid-level Intermediate Representation (MIR), which contains rich type information that we can take advantage of. The bug detection procedure considers the common patterns that we observe from existing reported bugs. To demonstrate the capability of MirChecker, we test our tool on 10 buggy code snippets extracted from existing bugs and more than 1,000 real-world Rust crates. Our evaluation shows that MirChecker is able to detect all the issues in our artificial code snippet dataset, and in particular, it finds 33 previously unknown bugs including 16 memory-safety bugs in 12 real-world Rust crates. With the help of some false-positive suppression heuristics, in our experience, the manual effort of identifying bugs from generated diagnostic messages is acceptable.

To sum up, our contributions are listed as follows:

- We present a new bug detection tool that is dedicated to Rust programs. By utilizing static analysis and constraint solving techniques, MirChecker can generate diagnostic messages that assist developers to quickly pinpoint potential bugs in their programs.
- We propose to use a dedicated abstract domain that keeps track of both numerical and symbolic values in Rust programs. We show that this design captures the common pattern of Rust vulnerabilities, and it is suitable for analyzing the structure of Rust MIR.
- We implement MirChecker and evaluate its effectiveness and performance in both artificial dataset and real-world Rust codebases. MirChecker reveals 33 previously unknown bugs including 16 memory-safety issues. We will open-source our system and dataset (§ 13) which may become the basis of other research in the future.

The rest of the paper is organized as follows. In Section 2 we give the background knowledge of static analysis and the Rust programming language. In Section 3 we briefly introduce the existing bugs found in Rust. The design of MirChecker is presented in Section 4, and we illustrate how Abstract Interpretation is performed on our language model in Section 5. The algorithms and implementation for processing the control flow graph and generating security conditions is discussed in Section 6 and Section 7. Finally, we report the experiments we have done and show the evaluation results in Section 8.

2 BACKGROUND

In this section, we introduce some preliminaries of static bug-finding techniques and the Rust programming language, especially its novel ownership-based resource management model, how it provides security guarantees, and why programs written in Rust may still have bugs.

2.1 Static Analysis and Bug-Finding

Conventional bug-finding approaches such as manual code review and unit testing are neither scalable enough to deal with sophisticated software with enormous amounts of code, nor convincing enough to prove the absence of vulnerabilities. Among all approaches, static analysis is attractive for its following features. First, static analysis is automated so it saves a lot of human effort. Second, it statically detects bugs without running a program, so it does not introduce extra runtime overhead. Finally, it can be used before the program is deployed, preventing severe consequences caused by critical bugs in advance. Many successful static bug-finding techniques and frameworks have been used in the industry, such as pattern matching [34], program inconsistency [27, 28], dataflow analysis [36] and symbolic execution [37].

In general, static program analysis is an automated technique that can extract runtime properties of programs without running them. Unfortunately, according to Rice's theorem [49], any non-trivial semantic property of programs is undecidable, which means perfectly precise static analysis is fundamentally impossible. *Abstract Interpretation* [18–20] addresses this problem by maintaining a sound over-approximation of the execution state at every program point. It models the program execution in a certain *abstract domain*, and each element of the domain represents a certain program state, which is referred to as an *abstract state*. Classical literature usually represents an *abstract state* as a *lattice* (a partially ordered set

$(State, \sqsubseteq)$ with a *join* operator $\sqcup$), while the statements of the program semantics are modeled as *abstract transfer functions* over *abstract states*. For example, interval analysis [18] computes for every integer variable a lower bound and an upper bound for its possible values. The *abstract state* is a lattice which consists of infinite number of intervals $[l, h]$. The order relation $\sqsubseteq$ is defined as interval inclusion, i.e., the order of $s_1 = [l_1, h_1]$ and $s_2 = [l_2, h_2]$ is defined as $s_1 \sqsubseteq s_2 \Leftrightarrow l_1 \geq l_2 \wedge h_1 \leq h_2$. The *join* operator $\sqcup$ is defined as the union of two intervals, i.e., $s_1 \sqcup s_2 = [min(l_1, l_2), max(h_1, h_2)]$. Special elements $\top$ (top) and $\bot$ (bottom) are defined to represent the greatest and the least element in the lattice, respectively. During the analysis, each integer variable is assigned with an interval which represents a sound approximation of its possible values.

The analysis is performed on the program's control flow graph (CFG), which contains information about the dependency relations between each basic block. Each statement is associated with a *transfer function*, which takes as input an *abstract state* and outputs a new state.

The design of abstract domains and transfer functions highly depends on applications. Numerous abstract domains are proposed in order to capture different properties. For example, numerical abstract domains such as interval [18], octagon [43], polyhedra [21], and congruence [32] approximate the numerical value of each variable. Some specialized abstract domains are designed for specific tasks, such as pointer analysis and cache-based side channels detection.

2.2 The Rust Programming Language

Rust is known for its ability to build programs that are both fast and secure. The Rust type system rigorously enforces strict disciplines to eliminate security issues. The most unique feature that distinguishes Rust is the *ownership* system, which enables Rust to guarantee memory safety without performing garbage collection.

The main idea of the ownership system is derived from concepts of *linear logic* [31] and *linear types* [52], which mean all values must be used exactly once. However, linear type systems are too restrictive for real-life applications. Therefore, Rust uses the concept of *ownership*, which relaxes the constraint of pure linearity. Under the ownership system, each value (e.g., an integer on the stack) has a unique *owner* (the variable binding). The scope of an owner determines the lifetime of its value. Ownership can be *moved* (transferred) between owners. Once the ownership is *moved*, the value is no longer accessible from the original variable binding. Rust also supports *references* that temporarily *borrow* a value from the owner without invalidating it. References are either *mutable* or *immutable*. The regulation is "no mutable aliasing", meaning that it is safe to have more than one reference to a value as long as the value is read-only; when the value is writable, only one single reference is allowed to exist. The restrictive rules for *move* and *borrow* are the key to achieving memory safety. On the one hand, the lifetime of each value is kept track of by its owner, and therefore the lifetime of a reference cannot exceed the value it points to. This effectively prevents many memory safety issues caused by dangling pointers. On the other hand, neither reference counting nor garbage collection is needed because resources are bounded with the lifetime of objects. The Rust compiler can automatically deallocate resources once their owners go out of their scopes. Note that all of the above are done at compile time thus no runtime overhead is introduced.

Although Rust has made significant progress in achieving safe system programming, it also provides an `unsafe` keyword which may breach the safety promise and lead to undefined behaviors.

3 MOTIVATION

Before we elaborate on the details of MirChecker, we first give an overview of the existing vulnerabilities of Rust. Specifically, we illustrate our observation which motivates the design of MirChecker. Then we give two examples to illustrate why Rust's type safety guarantee is not sufficient for security-critical scenarios, and how static analysis may provide mitigation.

3.1 Rust Bugs Overview

Numerous empirical studies [29, 42, 47, 53, 54] have shown that despite Rust's security features, vulnerabilities still exist in real world. By carefully inspecting the existing vulnerabilities, we focus on two categories of bugs:

Runtime Panics. In general, Rust's type system cannot enforce all the security conditions at compile time. Therefore, some conditions such as array bounds checking and integer overflow detection are postponed until runtime. The compiler automatically instruments assertions that would abort the execution if a security condition is violated. We observe that incorrectly manipulated integer values cause most runtime panics. For example, arithmetic overflow, out-of-bounds indexing and division by zero are all related to integers. Although aborting execution prevents memory corruption, it still causes denial-of-service attacks. According to a third-party bug collection repository `trophy-case`[1], about 40% of bugs are categorized as arithmetic error or out-of-range access.

Lifetime Corruption. Traditional memory corruption issues like use-after-free and double-free are well addressed by Rust's ownership system in the safe realm. However, the side-effects of the ownership system and the capability of `unsafe` together lead to a new pattern of dangling-pointer issues: Unsafe code may first corrupt the ownership system, causing invalid pointers or shared mutable aliases, then the ownership system automatically drops memory which further leads to double-free or use-after-free errors. This kind of memory corruption pattern is first reported by Xu *et al.* [22, 53]. In this paper, we will name it as *lifetime corruption*.

Based on the observations, we aim to combine numerical and symbolic static analysis, as they can capture both the paradigms of the above two kinds of errors. First, we use numerical static analysis to get bounds for each integer variable, such that potential runtime panics caused by integer-related operations can be detected. Second, we use symbolic static analysis to keep track of the ownership of heap memory. Especially, we focus on several unsafe functions that may create aliases of heap memory, as lifetime corruptions are usually related to shared mutable aliases and dropping heap memory.

[1] https://github.com/rust-fuzz/trophy-case

3.2 Motivating Examples

We give two existing vulnerabilities found in Rust, an integer overflow and a use-after-free, which respectively lie in the two categories we mentioned above. Unrelated details have been removed for simplicity of presentation. Listing 1 shows the integer overflow (CVE-2017-1000430 [10]) discovered in the `rust-base64` crate.

```
fn encoded_size(bytes_len: usize, config: Config) -> usize {
  let rem = bytes_len % 3;

  let complete_input_chunks = bytes_len / 3;
  let complete_output_chars = complete_input_chunks * 4;
  let printing_output_chars = if rem == 0 {
      complete_output_chars
  } else {
      complete_output_chars + 4
  };
  ...
}
```

Listing 1: Function `encoded_size` could overflow `usize` and eventually lead to buffer overflow.

The vulnerable function `encoded_size` takes an integer `bytes_len` as input. This integer is first divided by 3 and then multiplied by 4, and this computation may cause an integer overflow. This function is used to calculate the size of a heap buffer that needs to be reserved, and thus eventually leads to a smaller buffer. Accessing this buffer may result in a buffer over-read.

Numerical static analysis is able to detect this kind of vulnerabilities because the possible values of each variable are computed. For example, a static analyzer with interval abstract domain may reason about this code snippet as follows. First, according to the type of the input integer, the range of `bytes_len` can be represented as an interval $[0, 2^{64} - 1]$ (we assume a 64-bit platform). Then, according to the interval arithmetic, after the division and multiplication, the range becomes $[0, (2^{64} - 1)/3 \times 4]$. Finally, depending on the value of `rem`, the possible outcome is either $[0, (2^{64} - 1)/3 \times 4]$ or $[4, (2^{64} - 1)/3 \times 4 + 4]$. By taking the least upper bound of these two intervals, we obtain an over-approximation $[0, (2^{64} - 1)/3 \times 4 + 4]$, which is larger than the valid range of type `usize`. A static analyzer can help to identify this potential integer overflow.

```
fn from(buffer: Buffer) -> Vec<u8> {
    let mut slice = Buffer::allocate(buffer.len);
    let len = buffer.copy_to(&mut slice);
    unsafe {
        Vec::from_raw_parts(
            slice.as_mut_ptr(), len, slice.len())
    }
}
```

Listing 2: Mutable aliases created by `unsafe` could cause a use-after-free.

Our second example shows a use-after-free (CVE-2019-16140 [13]) found in the `chttp` crate, as shown in Listing 2. The `from` function first allocates a buffer `slice` and returns a vector composed by the unsafe function `Vec::from_raw_parts`, which obtains the ownership of `slice`. This unsafe behavior breaks the consistency of the ownership system, i.e., a chunk of memory now has two owners simultaneously. When the function `from` returns, the automatic dropping of `slice` deallocates the memory thus makes the returned vector invalid, causing a use-after-free when accessing the return value later.

Several existing use-after-free bugs (e.g. CVE-2019-15552 [11] and CVE-2019-15553 [12]) are due to the similar reason: the unsafe code corrupts the ownership system, then the automatic dropping mechanism causes dangling pointers. While this kind of problems is difficult to be detected by manual inspection, static analysis can help. The idea is to analyze the *control flow graph* (CFG) of a program and symbolically keep track of the ownership of each allocation. In the above example, by analyzing the ownership transitions, it is possible for a static analyzer to see that both the return value and `slice` point to the same chunk of memory. Then the memory deallocation statement (`Drop`) followed by the return statement (`Return`) corrupts memory.

4 DESIGN

In this section, we first illustrate the design choices of our static analysis methodology and their advantages. Then we present the high-level architecture of MirChecker.

4.1 Methodology

In general, performing static analysis requires modeling program semantics. We propose to combine numerical static analysis and symbolic execution, and perform analysis on top of Rust MIR.

In contrast to leveraging existing efforts on static program analysis such as IKOS [7], Crab [14] and KLEE [8], which perform static analysis on either LLVM [38] bitcode or self-defined intermediate representation, we decide to develop MirChecker from scratch based on Rust MIR for the following reasons. First, Rust MIR reduces most of the complex syntax of Rust into a much simpler core language, more importantly, it preserves type information and debugging data that we can take advantage of. For static analyzers, in general, the more information they have, the easier it is to find errors and suppress false positives. Second, although Rust uses LLVM as its backend, we find that most existing tools are dedicated to C/C++ and cannot directly work for Rust. Possible problems span from the lack of support of certain LLVM IR patterns to the lack of suitable standard library models [30]. Third, off-the-shelf tools are usually only compatible with specific versions of LLVM, thus users may have to compile their code using outdated Rust compilers in order to meet the requirements.

We perform integer bounds analysis using numerical abstract domains based on the observation that most runtime panics are caused by integer-related issues such as arithmetic errors and out-of-bounds access. Especially, numerical abstract domains are suitable for modeling low-level security-critical Rust programs, like embedded systems, where integer manipulations are usually heavily used. While performing numerical analysis, we construct symbolic formulas according to MIR's data structures. We also define a set of syntax-driven reduction rules that symbolically evaluate the formulas whenever possible (§ 5.2). As we will show, the combination of numerical and symbolic analysis mutually improves each other. On the one hand, Rust MIR contains complicated structures like references and arrays, which are problematic for numerical abstract domains, as they usually only support arithmetic operations and do

not understand the underlying memory layouts. Therefore, we utilize symbolic evaluation to work as the memory model and improve the precision of numerical analysis. On the other hand, the computational complexity of computing numerical abstract domains is usually much lower than SMT solving. Therefore our approach is more scalable than pure symbolic execution, where symbolic formulas may become too long and storing such information and solving constraints eventually become infeasible because of path explosion.

4.2 Architecture

The whole analysis process follows the canonical three-phase design of static analyzers: (1) user interface, (2) static analyzer and (3) bug detection, as depicted in Figure 1.

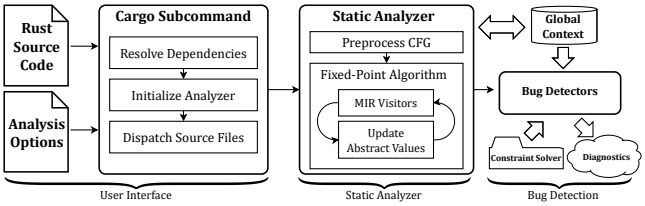

Figure 1: The architecture of MirChecker.

4.2.1 User Interface. We aim to provide a straightforward user interface such that users can use MirChecker together with other tools in the Rust ecosystem with minimal effort. Our user interface is a customized subcommand of `Cargo`, the official package manager of Rust. It reads a Rust crate as input, together with user-provided options that configure the behaviors of the analysis procedure (e.g., the entry point of the analysis). `Cargo` is called to automatically resolve dependencies of the input crate and collect all the needed source files. Then according to the user-provided options, an instance of static analyzer with appropriate configurations is initialized. We utilize the dependency information to dispatch source files, i.e., we only invoke the static analyzer on source files that are in the current Rust crate being analyzed, and use the vanilla Rust compiler to compile the remaining dependencies. This way, the amount of expensive analysis can be significantly reduced and only diagnostic messages for the targeting crate will be emitted.

4.2.2 Static Analyzer. Behind the user interface, the actual static analyzer is implemented as a modified Rust compiler with an additional analysis pass. The design goal is to extract both numerical and symbolic information from a *control-flow graph* (CFG). The analysis procedure is inserted as a customized callback function for the Rust compiler, which is invoked automatically after the compiler gathers all the information of the source code, thus we are able to access the internal compiler data structures and perform static analysis. The static analysis procedure first preprocesses the CFG for each function generated by the compiler and creates a *weak topological ordering* (WTO) [6] of the basic blocks (§ 6.1). Then according to the ordering, it adopts a fixed-point algorithm that iteratively executes *Abstract Interpretation* (§ 5) for each basic block and updates the result until it reaches a fixed point.

In the meantime, the global context works as an in-memory database for two purposes: (1) It stores the analysis results during the analysis phase, and further provides them to the bug detectors in the bug detection phase. (2) It caches data that is repetitively used during the analysis (e.g., the WTO for each function) in order to avoid redundant computation.

4.2.3 Bug Detection. After the analysis finishes, several bug detectors are invoked to detect potential vulnerabilities based on the analysis result, and diagnostic messages are generated accordingly. The bug detectors use constraint solving techniques to determine whether security conditions are violated. We consider two categories of security conditions: (1) runtime assertions and (2) common memory-safety error patterns. First, the Rust compiler automatically generates runtime assertions for security conditions that cannot be checked at compile time (e.g., out-of-bounds accesses and integer overflows). We utilize these assertions as verification conditions. The violation of this kind of conditions can trigger runtime panics but should not cause memory corruption. Second, we detect potential memory-safety issues based on the observations of the common pattern of *lifetime corruptions* (§ 3.1).

Finally, the diagnostics emission mechanism leverages the Rust compiler's internal infrastructures, therefore the diagnostic messages are structured and informative. The produced diagnostics show what kind of error might occur and its location. The expressive diagnostics can help users to pinpoint potential bugs in their programs quickly.

5 ABSTRACT INTERPRETATION

In this section, we introduce how our abstract domain is designed to capture both numerical and symbolic values during the analysis, and in particular, how they mutually improve each other. We define our minimal language model and memory model for Rust MIR. Based on these models, we introduce our abstract domain and illustrate how numerical and symbolic information can be extracted.

5.1 Language Model

We introduce a simple language that captures the core syntax of Rust MIR. The purpose of it is twofold. First, on top of this language model, we can define *abstract domains* and *transfer functions*. Second, it is used to construct symbolic expressions, as MirChecker's memory model (§ 5.2) symbolically evaluates these expressions to mimic memory accesses. Branch conditions are also stored as symbolic expressions and used for refining the analysis result (§ 6.1). Due to the complex nature of Rust, it is not realistic to model all the language features, thus we only extract necessary components useful for our purposes.

A Rust program consists of several functions and global variables. MirChecer uses a unified way to deal with both of them because the initialization procedure of each global variable can be viewed as a function as well. Each function can be represented as a *control flow graph* (CFG), where each node is a basic block containing one or more *statements* without any jumps. At the end of each basic block, there is one *terminator*, which is a special statement that represents a jump (either conditional or unconditional) among the control flow. We model the core syntax of the statements in *Backus-Naur Form* (BNF) as shown in Figure 2. For the convenience of

our implementation, the syntax mainly captures the skeleton of Rust MIR. Since the design principle of MirChecker is to apply numerical static analysis to Rust programs, we significantly simplify the basic data types that we care about: only integers (including Boolean and signed/unsigned integer data types of various lengths).

Constant	c	$\in$	$\mathbb{Z}$
BasicBlock	b	$\in$	$\mathbb{Z}$
Type	τ	::=	`Bool` | `I8` | `U8` | `I16` | `U16` | ...
Local	v	$\in$	$\{v_0, v_1, v_2, \ldots\}$
Function	f	$\in$	$\{f_1, f_2, f_3, \ldots\}$
BinOp	$\oplus$	::=	+ | - | $\times$ | $\div$ | % | `AND` | `OR` | `XOR` | `SHIFT`
CmpOp	$\otimes$	::=	< | $\leq$ | == | > | $\geq$ | $\neq$
Operand	op	::=	c | p
Place	p	::=	v | $*p$ | $p.n$ | $p[v]$
Rvalue	r	::=	op | $\&p$ |$!op$ | $-op$ | $op_1 \oplus op_2$ | $op_1 \otimes op_2$ | op `as` τ
Statement	s	::=	$s_1; s_2$ | $p = r$
Terminator	t	::=	`Call`$(f, [op_1, op_2, \ldots], (p, b))$ | `Drop`(p) | `Assert`(op) | `Goto`(b) `SwitchInt`$(op, [b_1, b_2, \ldots])$

Figure 2: Core syntax of the Rust language model.

We list the main syntax and its semantics that we are interested in most as follows:

(1) Assignment ($p = r$) overwrites the value in p with a new value expressed by the expression r. The left-hand side p is a **Place**, which can be either a single local variable **Local** or a qualified path representing accessing a field ($p.n$), dereferencing a pointer ($*p$), or indexing an array ($p[v]$). The right-hand side r is an **Rvalue**, which expresses operations performed on operands such as arithmetic ($op_1 \oplus op_2$), comparison ($op_1 \otimes op_2$), logical inversion ($!op$), negation ($-op$), getting reference ($\&p$) and type conversion (op `as` τ). Note that Rust MIR intentionally distinguishes **Place** and **Rvalue** to prevent **Rvalue**s from being nested in one another. The type system of Rust guarantees that p and r must be of the same data type.

(2) Binary operation ($op_1 \oplus op_2$) applies the binary operation $\oplus$ to two operands op_1 and op_2. Each operand can be a **Constant**, or a **Place**. The binary operator $\oplus$ includes normal arithmetic operators such as addition and subtraction. Therefore the data type of the two operands and the result are all integers.

(3) Comparison operation ($op_1 \otimes op_2$) similarly applies the comparison operation $\otimes$ to its two operands. It differs from binary operations in that it generates a Boolean value instead of an integer value. The results of comparison operations are usually used as branch conditions that may change the flow of control.

(4) Function call (`Call`$(f, [op_1, op_2, \ldots], (p, b))$) calls a function f with a list of arguments $[op_1, op_2, \ldots]$. The return value is assigned to p in basic block b. This is also used to allocate heap memory.

(5) Drop (`Drop`(p)) explicitly deallocates the memory of p which is allocated at runtime.

(6) Assertion (`Assert`(op)) continues execution if the condition stored in op is true, otherwise it triggers a runtime panic.

(7) Goto (`Goto`(b)) unconditionally jumps to the successor basic block b.

(8) SwitchInt (`SwitchInt`$(op, [b_1, b_2, \ldots])$) is a conditional branch instruction which transfers control flow to one of the basic blocks in the target list $[b_1, b_2, \ldots]$. At runtime, the discriminant operand op is evaluated into an integer value, and the control flow jumps to the basic block found by indexing the target list using this value. The last element in the target list is used for the default branch.

5.2 Memory Model

When analyzing a statement with memory accesses such as dereferencing pointers, a memory model is needed to describe how a static analyzer should handle these memory operations. For example, a statement `a[0] = *p` reads memory by dereferencing a pointer `p`, and writes the result to an array element. However, deciding which memory cell is accessed when encountering a memory read/write operation is notoriously difficult for static analysis, because a memory address may depend on the program's input and is usually only determined at runtime, therefore it is impossible for a static analyzer to know it at runtime. Traditionally, static analysis handles memory accesses with the assistance of a sound points-to analysis [44]. However, points-to analysis is usually used on low-level intermediate representations with a simple load/store memory model, where complex language abstractions have been lowered down into simple primitives such as reading/writing registers. On the contrary, Rust MIR has a much more complicated memory access paradigm than just load and store: it does not view memory as a single flat region. Instead, memory is treated as structured objects, each of which is identified using a **Place** expression, which contains a base variable and a list of projection elements (e.g., index, dereference) that "project out" from the base variable.

We therefore aim to implement a simple but rigorous memory model by leveraging the symbolic values that we construct during the analysis. When accessing a **Place**, we construct a symbolic expression for it and use it as an *abstract memory address*, i.e., we use the expression as the key of the memory lookup table. We determine whether two *abstract memory addresses* are equivalent by syntactically comparing the equality of the two expressions. This design may lose considerable precision because we may incorrectly regard two equivalent expressions as inequivalent due to the symbolic alias issue [4]. However, as we will show, we mitigate this problem by simplifying symbolic expressions using a set of reduction rules (§ 5.5).

5.3 Abstract Values and Abstract Domain

Let *CFG* be a given control-flow graph that is being analyzed, we define **P** to be the set of all **Place** occurring in *CFG*. To keep track of all the abstract values for each variable, we maintain a lookup table $\sigma_b : \mathbf{P} \mapsto \mathbf{V}$ for each basic block b, where **V** is the set representing all possible abstract values that a **Place** can be assigned to. We also

define two special values $\perp \in \mathbf{V}$, meaning an uninitialized value and $\top \in \mathbf{V}$, meaning all possible values.

The abstract values in **V** comprise of two disjoint categories: numerical values **NV** and symbolic values **SV**. Numerical values are used to capture the integer bounds of each variable, while symbolic values are constructed mainly to express abstract memory addresses (§ 5.2) and branch conditions (§ 6.1).

- Numerical abstract values in **NV** can be any valid integer constraint representation definitions used in classical Abstract Interpretation literature, such as intervals, octagons, and polyhedra. The goal is to abstractly bound numerical values for each variable, so that the constraints on integer bounds can provide a sound approximation of the program execution. We omit the definition here and refer the readers to any standard texts about Abstract Interpretation [18–20].
- Symbolic abstract values in **SV** represent a set of possible values on the right-hand side of an assignment. Therefore the syntax of symbolic abstract values are defined according to the `Rvalue` in the language model as stated in Figure 2. We store the symbolic values because many manipulations in Rust MIR cannot be easily modeled as integer bounds constraints, e.g., references, pairs, and indices. More specifically, when analyzing an assignment statement $p = r$ in basic block b, if the assigned value is not an integer, then we update the *program state* $\sigma_b[p] = \sigma_b[p] \cup r$. Hence for each memory path p, there is a set of corresponding possible symbolic values.

We define abstract state **AS** as a *map lattice* [44] consisting of the set of mappings from **P** to **V**. Similarly, Our abstract domain **AD** is defined as a *map lattice* consisting of mappings from **B** to **AS**, where **B** is the set of all the basic blocks in *CFG*. Intuitively, an element in **AS** is a lookup table, which maps from variables to abstract values, depicting the current execution state of the program. The abstract domain **AD** maintains a lookup table for each basic block, denoting different abstract states at the program point immediately after each basic block. The design of our abstract domain contains both numerical and symbolic values. In this paper, we informally call the numerical part of the abstract domain as numerical abstract domain, and the symbolic part as symbolic abstract domain.

Based on the above definitions, we also define *transfer functions*, which model each statement as an abstract state transformer. The input to the transfer functions represents the abstract state at the program point immediately before the statement, and the output represents the abstract state at the program point immediately after the statement. The behaviors of the transfer functions follow the above description of the language semantics in Section 5.1. For example, when analyzing an assignment statement, the transfer function looks up the abstract value at the left-hand side and updates it with the right-hand side. The transfer functions update abstract state according to either numerical or symbolic information from each statement. We illustrate how the information is extracted in Section 5.4 and Section 5.5.

An example given in Figure 3 shows the abstract values in the first iteration of the analysis. Figure 3 (a) is the compiled MIR of our example Rust code, which is simply a `for` loop with constant loop bound. Figure 3 (b) and (c) show the corresponding numerical abstract values and symbolic abstract values respectively. As shown in the figure, MirChecker's abstract domain models the execution state at the end of each basic block using two disjoint set.

5.4 Numerical Analysis

When the transfer functions analyze each statement, the numerical values are extracted and stored in the numerical abstract domain. The distinction between numerical values and others comes naturally from the different semantics of different operations. For example, computing the addition of two variables is obviously a numerical operation, while taking a reference is not. Our numerical abstract domain implementation (§ 7.1) mainly supports the following operations:

$$\begin{array}{llll} dst & ::= & src & \text{(Assignment)} \\ dst & ::= & op_1 \oplus op_2 & \text{(Binary arithmetic)} \\ dst & ::= & -op & \text{(Negation)} \end{array}$$

Therefore, when analyzing a statement, the transfer functions will first distinguish what kind of operations the statement does, and whether it should be handled numerically. More accurately, if a statement is an assignment, and the right-hand side is a variable of type integer, or a binary arithmetic operation or a negation, then the left-hand side will be handled by our numerical abstract domain. Otherwise, symbolic expressions are constructed and stored in the symbolic domain. In Figure 3, variable _1 and _5 are all handled by the numerical domain.

5.5 Symbolic Analysis

As mentioned above, Rust MIR still contains complex operations that cannot be easily modeled numerically. We therefore construct symbolic expressions and mainly use them for (1) constructing abstract memory addresses (§ 5.2), and (2) expressing branch conditions (§ 6.1). To address the symbolic alias issue [4], we define a set of reduction rules to simplify symbolic expressions as much as possible, as shown in Figure 4. We use Γ to denote the analysis environment, which is the current state of both numerical and symbolic domain. Notation $\Gamma \vdash [\![e]\!]$ means the evaluation of expression e in the environment Γ.

The reduction rules simplify symbolic expressions according to the language semantics. For example, rule Dref expresses the fact that dereferencing a reference to a variable is equivalent to reading the variable itself. This helps MirChecker to understand memory accesses. For example, in Figure 3, variable _2 is a reference to _1. At line 6, 14 and 18, dereferencing _2 results in variable _1, and thus appropriate abstract values can be updated. During the reduction procedure, information from numerical analysis is helpful, for example, rule Comp says if the comparison result can be derived from integer bounds analysis, then we can reduce the comparison expression into the Boolean result. If the simplified result is an integer, the value is moved into numerical abstract domain. For example, in Figure 3, the numerical values of variable _3 and _4 come from the symbolic reduction.

6 ALGORITHM

In this section, we discuss the essential algorithms of MirChecker and the rules for bug detection. Additionally, we show how to

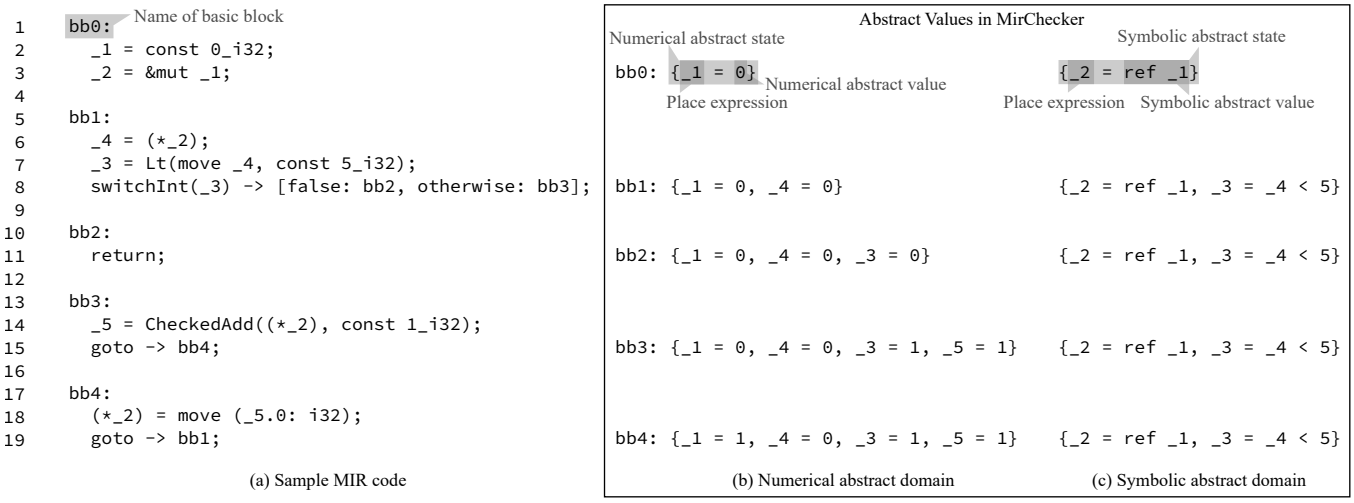

Figure 3: The abstract states in the first iteration.

$$\frac{r \in \textbf{Rvalue} \qquad p \in \textbf{Place} \qquad r = \&p}{*r \Rightarrow p}\ \textsc{Deref}$$

$$\frac{op_1, op_2 \in \textbf{Operand} \qquad b : \texttt{Bool} \qquad \Gamma \vdash [\![op_1 \otimes op_2]\!] = b}{op_1 \otimes op_2 \Rightarrow b}\ \textsc{Cmp}$$

$$\frac{p \in \textbf{Place} \qquad v \in \textbf{Local} \qquad c \in \textbf{Constant} \qquad \Gamma \vdash [\![p[v]]\!] = c}{p[v] \Rightarrow c}\ \textsc{Index}$$

$$\frac{p \in \textbf{Place} \qquad n \in \mathbb{Z} \qquad c \in \textbf{Constant} \qquad \Gamma \vdash [\![p.n]\!] = c}{p.n \Rightarrow c}\ \textsc{Field}$$

Figure 4: Reduction rules for symbolic expressions.

leverage the lifetime information encoded in MIR to clean up states for dead variables.

6.1 CFG Traversal

Similar to most of the existing static analysis tools, MirChecker traverses the CFG and iteratively runs static analysis until it reaches a fixed point. Inside the Rust compiler, the CFG for each function is encoded as a set of MIR data structures. It is a directed graph with nodes representing basic blocks and edges representing the control flow transitions. Note that a CFG contains the dependency relation between each basic block. Therefore, intuitively, the best traversing strategy is to follow the topological ordering of the CFG, such that the abstract value of a basic block only needs to be recomputed if its predecessors are updated. However, this method is not applicable since a CFG may have loops and thus the topological ordering is not well-defined. We tackle this problem by applying a classical strategy called *weak topological ordering* (WTO) [6], which is a generalized version of topological ordering applicable for all directed graphs.

The fixed-point algorithm is presented in Algorithm 1. The input CFG is first preprocessed by sorting all its basic blocks according to the *weak topological ordering*. The result is a list of topologically sorted *strongly connected components* (SCC) in the CFG. Each SCC is defined recursively as either a single basic block, meaning sequential execution, or a list of other SCC, indicating a loop. Then according to the ordering, an MIR visitor traverses through each SCC. For a sequential execution, the visitor simply visits each statement and updates the abstract values according to our language model (§ 5.1). For a loop, the MIR visitor traverses it repeatedly and will only proceed if a fixed point of this loop is reached. We also implement the standard *widening* and *narrowing* techniques [44], which guarantee that the fixed point can always be reached and thus our analyzer will not fall into infinite loops. Briefly speaking, if the number of fixed-point iterations exceeds a threshold, this technique will "widen" the variable to its maximum, then "narrow" it down for several iterations to get better precision. The default settings of both the widening threshold and the number of narrowing iterations can be changed through MirChecker's command-line options.

At the end of each basic block, there is a special statement called terminator which directs the control flow. Conditional branches are represented by a `SwitchInt` terminator. The `SwitchInt`(op, $[b_1, b_2, \ldots]$) terminator has two arguments: a discriminant operand and a target list. At runtime, op will be evaluated to an integer, which is used as the index to get a target basic block in the target list. Then the control flow jumps to the target. To exploit the information available in conditionals and make the numerical analysis more precise, we insert appropriate constraints to narrow the integer bounds. The constraints are generated by getting the symbolic values of op. For example, the control-flow transition of an if-else statement `if cond { . . . } else { . . . }` will be lowered into a terminator `SwitchInt`($cond$, $[b_1, b_2]$), which jumps to basic block b_1 if $cond$ is false, and to basic block b_2 otherwise. We therefore generate two constraints $cond == 0$ and $cond == 1$, and apply them to basic block b_1 and b_2 respectively. By utilizing the branch conditions, the numerical analysis can be

Algorithm 1: Fixed-point algorithm for MirChecker

Input: Control Flow Graph: *CFG*
Output: Abstract State: *State*
Init: $State[n] \leftarrow \begin{cases} \top & \text{if } n = \texttt{Entry}(CFG) \\ \bot & \text{otherwise} \end{cases}$

```
1  Function FixedPoint(CFG):
2      WTO ← ComputeWTO(CFG)
3      foreach scc ∈ WTO do
4          VisitSCC(scc)
5      return
6  Function VisitSCC(scc):
7      match scc with
8          node → VisitNode(node)
9          circle → VisitCircle(circle)
10     return
11 Function VisitNode(node):
12     pre_cond ← ⊔_{n∈Predecessors(node)} State[n]
13     AnalyzeBasicBlock(node, pre_cond)
14 Function VisitCircle(circle):
15     head ← circle.head
16     pre_cond ← ⊔_{n∈Predecessors(head)} State[n]
17     while true do
18         AnalyzeBasicBlock(head, pre_cond)
19         foreach scc ∈ circle.body do
20             VisitSCC(scc)
21         new_state ← ⊔_{n∈Predecessors(head)} State[n]
22         if new_state ⊑ State[head] then
23             break
24         else
25             pre_cond ← Widening(pre_cond, new_state)
26 Function AnalyzeBasicBlock(bb, pre_cond):
27     mir_visitor.pre_cond = pre_cond
28     foreach stmt ∈ bb.statements do
29         State[bb] ← Transfer(mir_visitor, stmt)
30     Transfer(mir_visitor, bb.terminator)
```

more precise, which further provides more information for the symbolic reduction procedure, as mentioned in Section 5.5.

6.2 Interprocedural Analysis

When the MIR visitor encounters a function call, the target function will only be analyzed if it is within the current crate that is being analyzed and its location can be statically determined. Otherwise, if the target function is from dependency crates or it is looked up at runtime (i.e., dynamic dispatch through vtable), the function call is skipped for simplicity. Note that this design obviously introduces unsoundness and makes our analysis result unreliable. However, thanks to Rust's capabilities of "zero-cost abstractions", in real-world Rust code bases, most functions are implemented using static dispatch for better performance. In addition, this design prevents MirChecker from analyzing unrelated code from external dependencies and avoids path explosion. Since the target function may be generic, the MIR visitor leverages the Rust compiler's internal API `GenericArg::expect_ty` to determine the real type of each argument according to the current context. After gathering all the information of function arguments, the target function is analyzed based on a new context, and the MIR visitor will handle the side-effects of the callee function and make appropriate updates in the caller's context.

We provide some special handlers for some functions that are commonly used but hard to be analyzed, especially for some standard library functions such as `index` and `as_mut_ptr`. These handlers work as the model of the corresponding functions by resembling the behaviors of them. MirChecker internally maintains a map between such special functions and their handlers, and will simply execute the handler instead of launching a new analysis if the functions are encountered.

Recursive functions may result in infinite loops during interprocedural analysis. MirChecker resolves infinite recursions by maintaining a list that simulates the call stack. Before analyzing each function, the function name is pushed into the list and popped out after the analysis of this function finishes. When the function being analyzed has already been in the list, MirChecker will directly return because a recursive call is detected. Although this approach introduces imprecision as recursive calls are skipped, it effectively improves the performance and guarantees the convergence of the analysis.

6.3 Verification Conditions for Bug Detectors

After the fixed-point algorithm finishes, we detect potential bugs based on the analysis result. The bug detectors leverage the information from both numerical and symbolic domain, and verify some security conditions through SMT solving. Diagnostic messages are produced if the SMT solver affirms that these security conditions can be potentially violated. We describe the security conditions that are verified by each kind of bug as follows.

Runtime Panics. The runtime panic detector traverses through the CFG and gets all the conditions from the `Assert(cond)` terminators. Then it translates the integer bounds from the numerical abstract domain and the symbolic values such as comparison expressions from the symbolic abstract domain into SMT formulas. An SMT solver take these SMT constraint formulas as input, and verifies the satisfiability of the negation of each condition *cond*. If satisfiable, meaning that the assertion condition is possible to be violated, the bug detector will generate diagnoses accordingly.

Lifetime Corruptions. To detect lifetime corruptions, MirChecker internally maintains a list of unsafe functions such as `Vec::from_raw_parts`. During the symbolic analysis, the transfer functions gather the ownership transitions made by these unsafe functions. For example, function `Vec::from_raw_parts` acquires the ownership from its first argument and returns an "owned" value with this ownership. The lifetime corruption bug detector verifies whether the original owner is used (e.g., used in a `Return` terminator) after the ownership has been transferred.

6.4 Eliminating Dead Variables

One advantage of using Rust MIR to perform static analysis is that the lifetime of each variable is explicitly encoded by

the `StorageLive` and `StorageDead` statements. Therefore MirChecker can safely clean up the storage of dead variables without performing *live variables analysis* [44], thereby achieving better performance. Cleaning dead variables is especially important for complex numerical abstract domains where the more variables they maintain, the more computational resources they consume. More specifically, if dead variable cleaning is enabled, whenever MirChecker encounters a `StorageDead` statement, it searches for the dead variable in both numerical and symbolic domains. If the variable is not depended by any other values, then its storage is cleared. As we will show in our evaluation (§ 8.3), eliminating dead variables reduces the analysis time and memory consumption for complex numerical abstract domains such as octagon and polyhedra.

7 IMPLEMENTATION

MirChecker is implemented in Rust (in 11,927 LOC) as a customized callback function on top of the official Rust compiler. It also cooperates with `Cargo`, the official package manager for Rust, providing a similar user experience to existing tools integrated with `Cargo`, such as the Rust linter `Clippy`. We implement our numerical abstract domain based on a third-party library called Apron [35], which provides a universal API for several numerical abstract domains including intervals [18], octagon [43] and polyhedra [21]. Since Apron is written in C, we develop a thin wrapper for the Apron API using Rust Foreign Function Interface (FFI) in order to use it in Rust. We also use GNU Multiple Precision Arithmetic Library (GMP) to handle arbitrary precision integers. The symbolic evaluation mechanism and the memory model is implemented based on MIRAI [15], a contract-based verification tool for Rust, which provides a general framework for traversing the data structures of Rust MIR. We achieve SMT solving by integrating the SMT solver Z3 [23] through the FFI binding of its C API. When launching bug detectors, the numerical constraints maintained by Apron and the symbolic constraints in the symbolic domain are translated into Z3 constraint formulas. Then the Z3 solver solves the constraints, and we accordingly generate diagnostic messages.

7.1 Binding External C APIs

In order to provide support for numerical abstract domains, we use a third-party library Apron [35]. To integrate it into our project, on the one hand, we need to implement FFI bindings for it because Apron is written in C. On the other hand, we need a "converter" to translate expressions from Rust to their equivalent form in Apron and vice versa. In Apron, each numerical abstract value is internally represented by a conjunction of a set of linear constraints on numerical properties of program variables: $\left\{\bigwedge_j \left(\sum_i \alpha_{ij} p_i \odot \beta_j\right)\right\}$, where $\alpha, \beta \in \mathbb{Z}$ are constants, $p \in$ **Place** is a symbolic path, and $\odot \in \{<, \leq, ==, \neq\}$ is a comparison operator. For example, statements `a=b+1; c=a+b;` will be stored as a linear constraint system $(a - b = 1) \wedge (a + b - c = 0)$. In MirChecker we define data structures to represent linear constraint systems. During the numerical analysis, MirChecker constructs linear constraints and converts them into the format that Apron understands. After the fixed-point algorithm terminates, the constraints are further translated into Z3 expressions so we can leverage the power of constraint solving provided by Z3.

Table 1: Datasets used in our experiments.

Dataset	# of crates
Code snippets extracted from existing bugs	10
Crates collected from crates.io	~1000
Crates searched on GitHub	25

7.2 Converting Abstract Values into SMT Formulas

As mentioned earlier, upon the termination of analyzing each function, we identify bugs by constructing SMT constraint formulas. The SMT solver Z3 is used to solve these constraints and check whether potential bugs may occur. A Z3 SMT formula is an abstract syntax tree (AST), where tree leaves are symbols or concrete data while other nodes are operators. The SMT constraint formulas are constructed from different sources: (1) The integer bounds computed by the numerical analysis. These constraints can be translated easily as they have already been represented by linear constraint systems. (2) The valid range of integer types. For example, a variable of type `u8` can hold values between 0 and 255. (3) The comparison expressions from the symbolic analysis. These expressions usually come from branch conditions and express the precondition of executing a basic block. All of these constraints are translated bottom-up into Z3 ASTs.

8 EVALUATION

We evaluate the effectiveness and performance of our work from two different perspectives. First, to evaluate MirChecker's ability of discovering currently known bugs, we test our tool on a synthetic dataset gathered from existing empirical studies on Rust security issues. To reflect the essence of each bug, we manually remove irrelevant code to make each test case "minimal". Our final dataset contains four memory-safety bugs and six runtime panic bugs. All of them can be successfully detected by MirChecker. Second, to evaluate MirChecker's bug-finding ability in real life, we collect real-world Rust crates on the official crate registry[2] based on the following requirements: (1) It contains code that performs computation instead of just macros or trait definitions. (2) It is unrelated to floating points operations, multi-threading, asynchronous programming, etc., since these are not our concern. We query the API of crates.io and collect around 1,000 crates that meet our requirements. We also collect crates by searching for the "`unsafe`" keywords on GitHub, because the improper use of `unsafe` is the primary source of memory-safety problems.

The datasets used in our experiments are listed in Table 1. All the experiments are done on a machine with a 3.70 GHz Intel Xeon E5-1630 v4 CPU and 16GB RAM, running Gentoo Linux (kernel 5.10.27).

8.1 Effectiveness

In our experiments, we evaluate more than 1,000 real-world Rust crates collected from both the official crate registry and GitHub. For each crate, we extract all the public functions and methods and use them as the entry functions of static analysis. In total, 17

[2] https://crates.io

Table 2: Evaluation result overview. The types of bugs include division-by-zero (DBZ), integer-overflow (IOV), out-of-range access (OOR), use-after-free (UAF), double-free (DF), and other panics not covered above (PANIC). Different numerical abstract domains are compared, including interval (ITV), linear congruence (LCG), octagon (OCT), and polyhedra (POL).

Crate Name	Bugs Confirmed	Warnings Reported (w or w/o false-positive suppression)				Elapsed Time (CPU seconds)				Peak Memory Usage (MB)				Bug Type	# of Lines	# of Entries	All-time Downloads*
		ITV	LCG	OCT	POL	ITV	LCG	OCT	POL	ITV	LCG	OCT	POL				
bitvec	1	26/70	23/66	22/59	22/59	1537.30	1520.37	1476.36	1472.79	322.91	323.98	345.27	326.58	DBZ	18139	359	5,722,435
brotli	3	433/633	411/598	416/605	416/605	10053.76	11315.82	10359.07	9530.12	501.43	502.66	923.61	578.20	IOV, OOR	108793	708	1,149,250
byte-unit	1	14/21	14/21	14/21	14/21	31.11	32.97	32.70	31.06	191.59	192.88	193.38	189.68	IOV	1420	45	850,943
bytemuck	2	10/15	10/15	10/13	10/13	13.20	12.84	11.18	10.56	156.00	157.40	157.39	157.20	PANIC	1320	29	3,389,482
executable-memory	2	2/2	2/2	2/2	2/2	27.90	27.36	27.19	26.95	189.70	190.38	191.18	190.46	IOV	160	15	1,092
gmath	15	57/57	67/67	67/67	67/67	24.92	163.83	351.74	155.54	126.47	129.77	188.66	133.68	UAF	292	17	N/A
qrcode-generator	1	14/15	14/14	14/14	14/14	247.31	827.33	825.70	825.91	290.47	290.14	290.73	290.88	IOV	722	42	24,201
r1cs	2	13/22	11/20	11/20	11/20	3008.19	2854.08	2727.46	2989.46	233.29	231.16	234.70	233.54	DBZ, OOR	4733	212	6,400
runes	2	107/219	103/212	106/218	106/218	385.51	730.68	754.09	679.79	163.27	163.47	621.36	360.71	IOV, DBZ	4180	188	5,114
safe-transmute	2	16/16	16/16	16/16	16/16	50.32	225.17	228.41	157.95	198.04	198.18	429.35	238.79	DBZ	1451	59	141,907
scriptful	1	3/7	3/7	3/7	3/7	19.37	18.91	19.16	19.13	155.51	156.54	156.89	156.39	PANIC	446	26	451
spglib	1	10/11	10/11	10/11	10/11	223.41	215.65	215.31	215.05	488.06	496.58	503.98	490.59	DF	441	14	471

* Sources: https://crates.io, as of September 13, 2021. N/A means the corresponding crate is not in the registry.

runtime panics and 16 memory-safety issues are detected in 12 crates. We list the detailed statistics in Table 2, where column "Bugs Confirmed" and "Warnings Reported" show the number of true positives we found and the number of warnings in the emitted diagnostic messages. After the diagnostic messages are generated, we manually inspect them to determine whether they are true positives or false positives. Since MirChecker outputs structured and informative diagnostic messages, identifying a true positive is relatively easy by reviewing the adjacent lines of code around the reported error site. Our experience is that an expert can inspect hundreds of diagnostics generated from 20 crates within 1 hour. Once we have confirmed a true positive, we write a simple code example that triggers the bug as a proof-of-concept (In Table 2, we only classify a warning as a true positive if we can really trigger it). All the bugs and the code examples were reported to the project maintainers. At the time of writing, 25 bugs in 7 crates were confirmed and 24 of them have been fixed by the maintainers.

8.2 False Positives Suppression

Most of the generated warnings are false positives. After carefully examining the diagnostic messages and the corresponding source code, we find the following reasons that cause the false alarms: (1) The nature of static analysis inevitably introduces imprecision, since the algorithm of MirChecker is to provide an over-approximation of program execution. (2) Many reported error sites can indeed be triggered. For example, some macros (e.g., `panic!()`, `unreachable!()`) in Rust cause a process to panic by design. But they are not considered as bugs since the authors intentionally implemented them. (3) A single bug may be triggered through different execution paths, resulting in multiple error sites being reported.

We thus implement some strategies to suppress the false positives. When MirChecker produces diagnoses, we record the root cause of each warning. E.g., arithmetic overflow, bitwise overflow, inline assembly, division-by-zero, running into panic code, etc. We classify all the diagnoses according to their cause, and provide command-line options for users to suppress some specific categories of warnings. For example, if a user is prototyping a program and temporarily uses many `panic!()` or `unreachable!()`. They may want to suppress all the warnings caused by running into panic code, so they can focus on detecting other errors.

In Table 2, we list the number of warning emitted with and without applying the false-positive suppression. In our experiments, we suppress warnings caused by bitwise overflow and running into panic code, because bitwise operations are not well-supported by Apron and thus MirChecker generates many false alarms. Note that in our evaluations, MirChecker performs poorly on crate `brotli` and `runes`. Even after applying our false-positive suppression strategies, MirChecker still performs a high false-positive rate at around 95.1%. However, if we exclude these two outliers, the false-positive rate reduces to around 79.2%.

8.3 Performance

We measure the analysis time and peak memory usage of MirChecker with different configurations, as shown in Table 2. We repeat our experiments using four different kinds of numerical abstract domains (interval, linear congruence, octagon, and polyhedra) and compare their precision and performance. As seen in Table 2, interval and linear congruence consume less resources but output relatively coarse-grained diagnostics thus more manual inspection is needed, while octagon and polyhedra are in contrast. Users can make a trade-off between computational efficiency and precision by choosing different options through MirChecker's user interface.

The elapsed time and memory consumption vary significantly among different test cases. We find that the main factor that affects the performance is the number of security conditions that MirChecker checks. Since MirChecker mainly checks numerical related conditions, in general, the more numerical operations a crate makes, the more resources its analysis consumes.

In addition, we test the performance of our dead variable cleaning mechanism (§ 6.4). We re-evaluate all the test cases in Table 2 when disabling dead variable cleaning, and compute the geometric mean for each abstract domain. We summarize both the results where dead variable cleaning is enabled (E) and disabled (D) in Figure 5. As one can observe, by taking advantage of the lifetime

information provided by the Rust type system, the dead variable cleaning mechanism in MirChecker reduces both the analysis time and memory consumption by 9.14% and 13.31% respectively for the octagon domain, and 16.89%, 11.03% respectively for the polyhedra domain. However, for interval and linear congruence, there is no significant difference in memory consumption. Also, the frequent cache cleaning operations introduce overhead so that there is only a slight improvement in execution time. We thus implement dead variable cleaning as optional, and users can choose whether it is enabled through MirChecker's user interface.

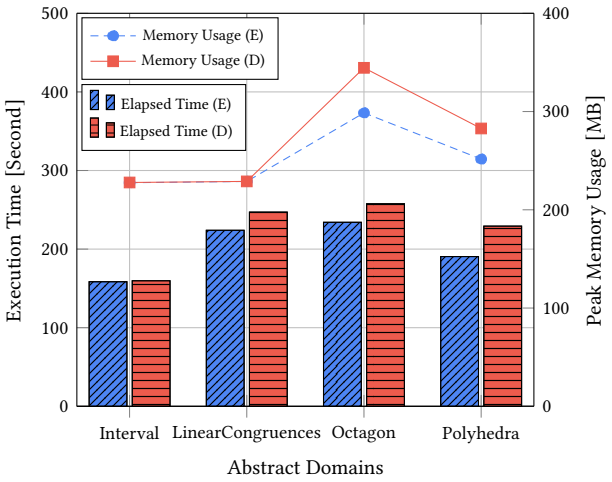

Figure 5: Performance of dead variable cleaning.

9 CASE STUDY

To provide a more concrete understanding of the effectiveness, we present two bugs detected by MirChecker, which respectively belong to *runtime panics* and *lifetime corruption* (§ 3.1). We also attach MirChecker's diagnostic messages in the comments to show how they help developers to pinpoint potential bugs quickly. For more examples, we refer the readers to our source code repository (§ 13).

Listing 3 gives an integer overflow which further leads to an out-of-bounds access. At line 17, a vector `img_raw` is allocated, and its length is computed at line 16. The computation of `length` is a multiplication of `size` which comes from the user input. Therefore, it is possible to construct malformed input and trigger an integer overflow at line 16, resulting in a smaller `img_raw`. Further manipulations of `img_raw` will cause an out-of-bounds access. As shown in the comments, MirChecker produces a warning which correctly detects the integer overflow, and pinpoints exactly where the problem is in a particular line of code. Note that for Rust, array bounds checking is always performed at runtime, therefore this bug only causes a runtime panic instead of leading to a memory-safety vulnerability.

Listing 4 gives a use-after-free vulnerability found during our experiments. First, a buffer `ptr` is allocated, then the ownership of `ptr` is transferred to a vector `mat` using `Vec::from_raw_parts`. When the function `matrix2invert` returns, `mat` is deallocated thus the

```
1  // The output of MirChecker:
2  // warning: [MirChecker] Possible error: attempt to compute
   ↪  `move _29 * move _30`, which would overflow
3  //    --> src/lib.rs:445:18
4  //     |
5  // 445 |     let length = size * size;
6  //     |                  ^^^^^^^^^^
7  fn to_image_inner(qr: QrCode, size: usize) -> Result<Vec<u8>,
   ↪  QRCodeError> {
8    // Fix by adding the following:
9    // if size >= 2usize.pow((size_of::<usize>() * 4) as u32) {
10   //     return Err(QRCodeError::ImageSizeTooLarge);
11   // }
12   let margin_size = 1;
13   let s = qr.size();
14   let data_length = s as usize;
15   let data_length_margin = data_length + 2 * margin_size;
16   let point_size = size / data_length_margin;
17   if point_size == 0 {
18       return Err(QRCodeError::ImageSizeTooSmall);
19   }
20   let margin = (size - (point_size * data_length)) / 2;
21   let length = size * size;
22   let mut img_raw: Vec<u8> = vec![255u8; length];
23   // Some manipulations on vector `img_raw`
24   // Skip for simplicity of presentation
25   // ... ...
26   Ok(img_raw)
27 }
```

Listing 3: An integer overflow and out-of-bounds access in crate `qrcode-generator`.

return value `ptr` becomes a dangling pointer. Further usage of `ptr` will cause a use-after-free. As shown in the comments, MirChecker detects this memory-safety issue, and pinpoints the variable whose memory is not properly managed.

```
1  // The output of MirChecker:
2  // warning: [MirChecker] Possible error: double-free or
   ↪  use-after-free
3  //   --> wasm/matrix2.rs:32:3
4  //    |
5  // 32 |   ptr
6  //    |   ^^^
7  pub unsafe fn matrix2invert(a: *mut f32) -> *mut u8 {
8    let a = std::slice::from_raw_parts(a, LEN);
9
10   let det = a[0] * a[3] - a[2] * a[1];
11
12   if det == 0.0 {
13     return std::ptr::null_mut();
14   }
15
16   let ptr = alloc(SIZE);
17   // Fix this by replacing line 19 with the following line:
18   // let mat = slice::from_raw_parts_mut(ptr as *mut f32, LEN);
19   let mut mat = Vec::from_raw_parts(ptr as *mut f32, LEN, LEN);
20   let det = 1f32 / det;
21
22   mat[0] = a[3] * det;
23   mat[1] = -a[1] * det;
24   mat[2] = -a[2] * det;
25   mat[3] = a[0] * det;
26
27   ptr
28 }
```

Listing 4: A use-after-free in crate `gmath`.

10 DISCUSSION

Amount and Severity of Bugs. In this work, most of the bugs detected by MirChecker are not memory-safety bugs. Instead, they trigger runtime panics and abort the execution. Not only is

the amount of unveiled bugs much smaller than many bug-finding efforts for C/C++, but most of the bugs also do not lead to damaging vulnerabilities. We argue that this is expected since Rust is memory-safe by default, therefore is reasonable that finding bugs in Rust programs requires more costs. This result coincides with other bug-finding techniques deployed in Rust, such as fuzzing [3], where most bugs detected only cause denial-of-service attacks. From the perspective of Rust developers, this is a promising result because it reflects the fact that the codebases in the Rust ecosystem are pretty safe, and exploitable memory-safety bugs are rarely seen.

Limitations and Future Work. While we believe our mechanism captures the common pattern of currently known Rust vulnerabilities, the main limitation of our tool is that it is not exhaustive. The memory model we use is lightweight and syntax-driven thus it cannot handle all the memory operations. Many advanced features like closures and higher-order functions are ignored for simplicity. Dynamic dispatch of methods, inline assembly, concurrency, and FFI are out of the ability of MirChecker. As a result, MirChecker does not prove the the absence of bugs, and may miss bugs because of the unsupported features. However, we argue that, on the one hand, our goal is to provide a useful bug detection tool rather than enforcing rigorous formal verification. The missing support for these features will not impede the execution of MirChecker, thus it can still produce useful diagnoses. On the other hand, since the structure of Rust MIR is complicated and subject to change, it requires too much engineering effort to be fully handled. Alternatively, we adopt our simple model and implement MirChecker as a handy tool for Rust developers. Note that further improvement and refinement of MirChecker are our future work.

11 RELATED WORK

Existing works on bug detection in Rust. Existing studies usually extend off-the-shelf static or dynamic analysis tools to perform bug detection on either Rust MIR or LLVM IR generated by the Rust compiler. For example, Lindner *et al.* [41] use the famous symbolic execution engine KLEE [8] to verify whether a program is panic-free. SMACK [48] is a translator from the LLVM IR into the Boogie intermediate verification language [24], and now it has been extended to support Rust [5]. Rust2Viper [33] and its successor Prusti [2] is a compiler plugin that utilizes user-provided specifications and a symbolic execution engine called Viper [45] to verify functional correctness properties. CRUST [50] is a test generation and model checking tool. It filters necessary functions that contain unsafe code and translates them into C code. The generated tests are checked by an off-the-shelf model checker CBMC [9]. Dewey *et al.* [26] propose a fuzzing technique to detect runtime execution crashes. Qin *et al.* [47] build two bug detectors for use-after-free and double-lock bugs according to their empirical studies on Rust security issues. SafeDrop [22] focuses on the deallocation of heap memory and detects memory corruptions by performing alias analysis and taint analysis on Rust MIR. Miri [16] is a Rust MIR interpreter that dynamically executes Rust code according to the operational semantics. When the interpreter gets stuck, it means there is an undefined behavior with respect to the operational semantics. Miri is designed for finding undefined behaviors instead of bug detection and has totally different functionalities from any static analyzers. MIRAI [15] is a formal verification tool that performs symbolic execution. It has a well-defined memory model for heap allocation. It enables users to add annotations and utilizes the SMT solver Z3 [23] to prove the safety of Rust programs.

Our work differs from the previous efforts in that we build our tool based on Rust MIR so we fully leverage the type information provided by the Rust compiler. Also, MirChecker detects a broader category of bugs than most of the previous works do, and it does not require manual annotations such that one can easily use it with minimal effort.

Empirical studies on Rust security issues. Empirical studies [29, 42, 47, 53, 54] on Rust security issues summarize some currently known bugs. Xu *et al.* [53] collect bug reports from public datasets which contain all existing CVEs of Rust, and manually analyze the root cause and taxonomy. Qin *et al.* [47] manually analyze hundreds of unsafe code usages in several open-source Rust projects, revealing the impacts of unsafe code on memory/concurrency safety. Evans *et al.* [29] statistically analyze the usage of unsafe code in real-world Rust libraries and applications and claim that although Rust provides a way to encapsulate unsafe code, the propagation of unsafeness becomes a big challenge.

12 CONCLUSION

In this paper, we presented MirChecker, a numerical static analysis tool based on a combined abstract domain, which captures both numerical and symbolic values and mutually improves each other. It assists Rust developers in detecting potential defects in their programs. We implemented MirChecker based on existing numerical abstract domains and constraint solvers and evaluated it by analyzing real-world Rust codebases. MirChecker successfully revealed 17 runtime panics and 16 memory-safety issues that were unknown previously. Finally, we open-sourced MirChecker with various examples and datasets.

13 AVAILABILITY

The source code, test scripts, and detailed information about bugs that we found are available online at:

https://github.com/lizhuohua/rust-mir-checker

ACKNOWLEDGMENTS

The work of John C.S. Lui is supported in part by the RGC R4032-18. We would also like to express our gratitude to our shepherd Prof. Erik van der Kouwe and anonymous reviewers for their constructive comments. Thanks to all the crate maintainers who responded to our bug reports.

REFERENCES

[1] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. 81–89.

[2] 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.

[3] Rust Fuzzing Authority. 2021. *Trophy Case.* https://github.com/rust-fuzz/trophy-case

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018), 1–39.

[5] Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. 2018. Verifying Rust Programs with SMACK. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA '18)*. 528–535.

[6] François Bourdoncle. 1993. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and their Applications*. 128–141.

[7] Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *International Conference on Software Engineering and Formal Methods (SEFM '14)*. 271–277.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. 209–224.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*. 168–176.

[10] CVE Contributors. 2017. *CVE-2017-1000430*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000430

[11] CVE Contributors. 2019. *CVE-2019-15552*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15552

[12] CVE Contributors. 2019. *CVE-2019-15553*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15553

[13] CVE Contributors. 2019. *CVE-2019-16140*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16140

[14] Crab Contributors. 2021. *CoRnucopia of ABstractions: a language-agnostic library for abstract interpretation*. https://github.com/seahorn/crab

[15] MIRAI Contributors. 2021. *MIRAI: Rust mid-level IR Abstract Interpreter*. https://github.com/facebookexperimental/MIRAI

[16] Miri Contributors. 2021. *Miri: An interpreter for Rust's mid-level intermediate representation*. https://github.com/rust-lang/miri

[17] RedoxOS Contributors. 2021. *Redox OS*. https://www.redox-os.org/

[18] Patrick Cousot and Radhia Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *Proceedings of the 2nd International Symposium on Programming (ISOP '76)*. 106–130.

[19] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. 238–252.

[20] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. 269–282.

[21] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '78)*. 84–96.

[22] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2021. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis. arXiv:2103.15420 [cs.PL]

[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. 337–340.

[24] Rob DeLine and Rustan Leino. 2005. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs*. Technical Report MSR-TR-2005-70. 13 pages.

[25] Rust For Linux Developers. 2021. *Rust for Linux*. https://github.com/Rust-for-Linux

[26] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. 482–493.

[27] Isil Dillig, Thomas Dillig, and Alex Aiken. 2007. Static Error Detection using Semantic Inconsistency Inference. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 435–445.

[28] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.

[29] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 246–257.

[30] Jack Garzella, Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. 2020. Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification. In *Proceedings of the 21st International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '20)*. 90–111.

[31] Jean-Yves Girard. 1995. Linear Logic: Its Syntax and Semantics. In *Proceedings of the Workshop on Advances in Linear Logic*. 1–42.

[32] Philippe Granger. 1989. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics* 30, 3–4 (1989), 165–190.

[33] Florian Hahn. 2016. *Rust2Viper: Building a Static Verifier for Rust*. Master's thesis. ETH Zürich.

[34] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *ACM SIGPLAN Notices* 39, 12 (2004), 92–106.

[35] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *International Conference on Computer Aided Verification (CAV '09)*. 661–667.

[36] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. 194–206.

[37] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. 75–86.

[39] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 234–251.

[40] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2019. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES '19)*. 1–10.

[41] M. Lindner, J. Aparicius, and P. Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN '18)*. 108–114.

[42] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing Unsafe Rust Programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 234–245.

[43] Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100.

[44] Anders Møller and Michael I. Schwartzbach. 2018. *Static Program Analysis*.

[45] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583 (VMCAI '16)*. 41–62.

[46] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[47] 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 Conference on Programming Language Design and Implementation (PLDI '20)*. 763–779.

[48] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV '14)*. 106–113.

[49] Henry Gordon Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[50] J. Toman, S. Pernsteiner, and E. Torlak. 2015. Crust: A Bounded Verifier for Rust. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. 75–80.

[51] Jeff Vander Stoep and Stephen Hines. 2021. *Rust in the Android platform*. https://security.googleblog.com/2021/04/rust-in-android-platform.html

[52] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*.

[53] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. arXiv:2003.03296 [cs.PL]

[54] Zeming Yu, Linhai Song, and Yiying Zhang. 2019. Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. arXiv:1902.01906 [cs.PL]