Broadly Enabling KLEE to Effortlessly Find Unrecoverable Errors in Rust

Ying Zhang[*]
yingzhang@vt.edu
Virginia Tech
Blacksburg, VA, USA

Peng Li[*]
peli@zoox.com
Zoox
Foster City, CA, USA

Yu Ding[*]
dingelish@google.com
Google
Mountain View, CA, USA

Wang Lingxiang[*]
lingxwang@microsoft.com
Microsoft
Redmond, WA, USA

Dan Williams
djwillia@vt.edu
Virginia Tech
Blacksburg, VA, USA

Na Meng
nm8247@vt.edu
Virginia Tech
Blacksburg, VA, USA

ABSTRACT

Rust is a general-purpose programming language designed for performance and safety. Unrecoverable errors (e.g., *Divide by Zero*) in Rust programs are critical, as they signal bad program states and terminate programs abruptly. Previous work has contributed to utilizing KLEE, a dynamic symbolic test engine, to verify the program would not panic. However, it is difficult for engineers who lack domain expertise to write test code correctly. Besides, the effectiveness of KLEE in finding panics in production Rust code has not been evaluated. We created an approach, called *PanicCheck*, to hide the complexity of verifying Rust programs with KLEE. Using *PanicCheck*, engineers only need to annotate the function-to-verify with `#[panic_check]`. The annotation guides *PanicCheck* to generate test code, compile the function together with tests, and execute KLEE for verification. After applying *PanicCheck* to 21 open-source and 2 closed-source projects, we found 61 test inputs that triggered panics; 59 of the 61 panics have been addressed by developers so far. Our research shows promising verification results by KLEE, while revealing technical challenges in using KLEE. Our experience will shed light on future practice and research in program verification.

ACM Reference Format:
Ying Zhang, Peng Li, Yu Ding, Wang Lingxiang, Dan Williams, and Na Meng. 2024. Broadly Enabling KLEE to Effortlessly Find Unrecoverable Errors in Rust. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639477.3639714

1 INTRODUCTION

Rust was created to ensure high performance comparable to that offered by C and C++, while emphasizing the code's safety—the Achilles heel of the other two languages [1]. Rust's error handling offers a robust and expressive mechanism that encourages developers to handle errors gracefully and explicitly.

Rust groups errors into two categories: recoverable and unrecoverable errors [2]. A **recoverable error** (e.g, *File Not Found*) is an error that does not cause the program to terminate abruptly. A program can retry the failed operation or specify alternative actions when it encounters a recoverable error [3]. For instance, if a Rust program attempts to open a file that does not exist, it is a recoverable error because the program can then proceed to create the file [4]. An **unrecoverable error** (e.g., *Index Out Of Bounds*) causes a program to fail abruptly. A program cannot revert to its normal state if an unrecoverable error occurs. It cannot retry the failed operation or undo the error. Namely, unrecoverable errors are symptoms of bugs, more dangerous than recoverable ones.

Most languages do not distinguish between these two kinds of errors; they handle both in the same way using mechanisms such as exceptions. Rust does not have exceptions [3]. Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro (a Rust macro is like a function) that stops execution when the program encounters an unrecoverable error. By checking whether the return-type of a Rust function is `Result<T, E>`, developers can easily identify recoverable errors, and implement code to eagerly handle those errors before compiling or running their software. However, it is much harder to identify unrecoverable errors. This is because such errors are not signaled by any dedicated return-type; developers have to reason about program semantics intensively to reveal errors. For simplicity, this paper uses **panics** to consistently refer to unrecoverable errors [5].

To explore potential panics in Rust code, prior work leverages symbolic execution to verify Rust programs. Specifically, they tried to compile Rust source code to LLVM bitcode, and used KLEE [6–8], a symbolic execution engine for LLVM, to symbolically execute Rust test code and uncover panics. For instance, Rust verification tools (RVT) [8, 9] is a collection of tools/libraries to support both random testing and verification of Rust programs. RVT provides libraries to patch the LLVM IR to support KLEE features like symbolic values. It requires users to manually define parametrized unit tests, compiles those tests together with Rust code into bitcode files, and automates the process of invoking KLEE on bitcode files.

However, it is challenging for developers who lack the domain expertise to write a parametrized unit test [10]. This specialized skill set requires an in-depth understanding of the function's intricacies, potential edge cases, and symbolic execution background to call that function in specialized ways [11]. Besides, even with

[*] Indicates authors who were previously employed by ByteDance Ltd.

ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0501-4/24/04.
https://doi.org/10.1145/3639477.3639714

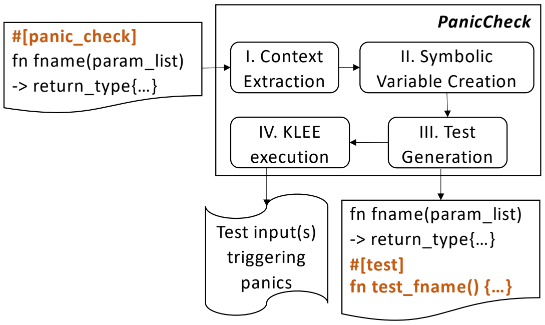

Figure 1: *PanicCheck* consists of four phases

RVT's help, it is still infeasible to do a large-scale evaluation due to too much human effort to configure and write tests for every project. This raises questions about its real-world applicability and reliability when used in Rust-based environments. As a result, many developers may hesitate to adopt symbolic execution until user-friendly tools emerge or case studies emerge that demonstrate its prowess in the Rust ecosystem.

To fill these gaps, we developed *PanicCheck*, a semi-push-button [12] dynamic test verification tool tailored for Rust. *PanicCheck* only requires engineers to annotate functions with `#[panic_check]`. It then handles compilation, test generation, symbolic execution, and panic checking automatically. This automation enables large-scale empirical evaluation of *PanicCheck* on Rust code. As shown in Figure 1, with *PanicCheck*, we can verify function `fname(...)` by annotating it with `#[panic_check]`. Given the annotated program, *PanicCheck* goes through four phases. Phase I compiles the program to extract the function name and parameters. Phase II determines how to create a symbolic variable for each parameter. Phase III creates and compiles a parametrized unit test to declare symbolic variables and call `fname(...)` with those variables. Phase IV executes the test with KLEE, to output any test inputs that trigger panics.

We evaluated *PanicCheck* and KLEE on real-world Rust projects, guided by two goals. First, we aimed to rigorously measure KLEE's effectiveness at finding panics in production Rust code. Second, we sought to identify areas for improvement in the symbolic execution workflow for Rust. We applied *PanicCheck* to 21 popular open-source Rust programs, and 2 large production-grade closed-source Rust programs that served as key infrastructures at ByteDance. We annotated hundreds of functions with `#[panic_check]`, and passed all annotated functions to *PanicCheck* for program verification. In total, *PanicCheck* revealed 61 panics in 6 of the projects. By examining developers' later changes to their projects, we found that 52 of the panics were fixed. Furthermore, we filed bug issues or pull requests for the remaining nine panics; so far, developers have fixed seven. The results provide new insights into KLEE's capabilities as well as guides future tool development.

We made the following contributions in this paper:

- We created a tool *PanicCheck*, which wraps the usage of KLEE and streamlines the verification process. Because little manual effort is required, *PanicCheck* enables us to conduct the large-scale case study, and helps us avoid human errors when creating unit tests.
- We conducted a case study by applying *PanicCheck* to 23 real-world Rust projects, in order to verify hundreds of Rust functions in those projects. No prior work conducts such a large-scale study as what we did.
- By observing the runtime behaviors of KLEE and analyzing all 61 panics it revealed in our study, we characterized the strengths and weaknesses presented by the tool.

In the following sections, we will first introduce the technical background KLEE and RVT (Section 2.1), and describe a running example (Section 2.3). Then we will explain *PanicCheck* (Section 3) and our experiment in detail (Section 4).

2 BACKGROUND AND MOTIVATION

In this section, we will first introduce the technical background of KLEE and RVT. Then we will describe a concrete scenario of Rust code verification to motivate our research.

2.1 KLEE

KLEE is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure [13], to automatically explore paths through a program and decide what inputs cause which part of the program to execute. Theoretically speaking, it can run any program compiled to LLVM bitcode. In practice, it has been mainly applied to C/C++ programs.

Given a function-to-verify (FTV), KLEE conducts inter-procedural analysis to explore various possible execution paths, and synthesizes the constraints on symbolic variables for explored paths. For each path, KLEE uses the constraint solver STP to solve the path condition, to decide whether that path is feasible. For each feasible path, KLEE generates a concrete input triggering the path, and checks if there are any values that can cause an unrecoverable error. KLEE is known to have the following limitations [14].

- Path explosion: The number of paths through a program can be exponential in the size of the program. Therefore, unless the program under analysis is small, KLEE cannot finish checking all possible paths in a timely manner and users need to set a timeout to terminate its execution.
- Bounded checks for loops: In general KLEE cannot show that a loop will always behave correctly. It only checks some of the possible executions of a loop.
- Long time spent in constraint solving: When some path conditions or constraints are hard to solve, STP may spend overly long time trying to find satisfying value assignments.

The kinds of bugs KLEE can find are memory errors (e.g., buffer overflows and null-pointer dereference), division/modulo by zero, over shifts, and assertion violations [15].

2.2 Rust Verification Tools (RVT)

RVT [8, 9] is a collection of tools and libraries to support both random testing and verification of Rust programs. It provides the functionalities to compile the Rust projects to LLVM bitcodes and invoke KLEE to verify the program against the LLVM bitcodes. To write a test with RVT, developers need to 1) define a test function to assert certain properties for their program (e.g., no panic will occur), 2) specify how each parameter should be symbolized using

RVT's domain-specific language (DSL) to generate the test inputs. Then, developers invoke RVT to compile the program into LLVM bitcode, synthesize the constraints on symbolic variables for each execution path, and decide whether the property always holds for the function-to-verify.

It can be challenging for developers to manually write test functions with RVT due to two reasons. First, the test functions involve the traits or APIs defined by RVT (e.g., abstract_value(...)), requiring that developers have sufficient domain knowledge of RVT and KLEE. Second, when a test function needs to prepare parameter inputs of complex data types (e.g., u8 slice reference), developers have to carefully prepare compound data structures (e.g., vector), by properly composing symbolic variables.

2.3 A Motivation Example

Lines 1–13 in Listing 1 show a function from a real-world Rust crate (i.e., compilation unit [16]): integer-encoding-rs-1.1.7 [17]. The function `decode_var(...)` takes in a variable of type `&[u8]` as input. Here, `[u8]` means **u8 slice**—a dynamically sized type representing a view into a contiguous sequence of elements of type u8 [18]; `&[u8]` refers to any reference to a variable of type u8 slice. Once an input is provided, the function decodes the input, and returns a tuple that includes (1) the decoded content and (2) a value of type usize (i.e., the pointer-sized unsigned integer type).

To verify the function together with all functions called by that function (e.g., `zigzag_decode(result)`) via RVT, developers need to manually craft a test function similar to the one shown by lines 16–28 of Listing 1. The demonstrated test code prepares a value of type `&[u8]`, calls `decode_var(...)` with that value, and checks whether any panic occurs. Specifically, to prepare the input parameter, the test code first declares a vector of u8, with an initial size set to 30 (see lines 18–19). Next, it defines 30 symbolic variables of type u8, by repetitively calling the function `u8::abstract_value()` (see lines 20–21). These symbolic variables are important for KLEE to later verify the program via symbolic execution. After declaring 30 symbolic variables and storing them into the vector variable `v` (see lines 20–23), the test code tentatively makes the call `decode_var(&v)`, where `&v` is a vector reference and is also of the type u8 slice reference.

In summary, when developers write a test, they need to:

- Identify the types of input parameters, no matter whether they are primitive, compound, or collection types.
- Write code to create symbolic variables of various types that KLEE can interpret. When a variable has a compound or collection type, decide on the size and create elements to put into the compound or collection variable.
- Write the code for verification and panic checking.

To save developers' manual effort and ensure the quality of program verification, we created *PanicCheck* — an automatic approach to generate test functions and conduct program verification using those test functions via KLEE. Given a function annotated with `#[panic_check]` (see Listing 2), PanicCheck generates a test function semantically equivalent to lines 16–28 in Listing 1.

Listing 1: A Rust program under testing and the test function manually defined for the usage of RVT's KLEE backend

```
1   //Decode a zigzag encoding value from the slice. Returns the value
    ↪  and the number of bytes read from the slice
2   fn decode_var(src: &[u8]) -> (Self, usize) {
3     let mut result: u64 = 0;
4     let mut shift = 0;
5     for b in src.iter() {
6       let msb_dropped = b & DROP_MSB;
7       result |= (msb_dropped as u64) << shift;
8       shift += 7;
9       if b & MSB == 0 || shift > (10 * 7) {
10        break;
11      }
12    }
13    (zigzag_decode(result) as Self, shift / 7 as usize)
14  }
15
16  // the test code that a developer needs to write if s/he wants to
    ↪  adopt the RVT's KLEE backend
17  #[test]
18  fn test_decode_var(){
19    let len = 30;
20    let mut v = Vec::with_capacity(len);
21    for _ in 0..len {
22      let element = u8::abstract_value();
23      v.push(element);
24    }
25    let result = panic::catch_unwind(|| {
26      let _ = decode_var(&v);
27    });
28    assert!(result.is_ok());
29  }
```

Listing 2: A Rust program annotated with `#[panic_check]`

```
#[panic_check]
fn decode_var(src: &[u8]) -> (Self, usize) { ... }
```

3 *PANICCHECK*

As described above, it can be tedious and error-prone for developers to manually write test functions, when they want to verify lots of Rust functions. To save developers' effort and ensure verification quality, we developed *PanicCheck*. Given a function annotated with `#[panic_check]`, *PanicCheck* parses that function to extract the function name, parameters, and their data types. It then generates a test function. The test function is compiled into LLVM bitcode, so that KLEE is applicable to verify `decode_var(...)`. The compilation process also injects value checks to guard critical instructions (e.g., arithmetic or bitwise operators), and adds `panic!` macros when value checks fail. *PanicCheck* streamlines the verification process by synthesizing tests for given Rust code, and invoking the existing toolchain in RVT for Rust-to-bitcode conversion as well as KLEE application.

As shown in Figure 1, *PanicCheck* defines Phases I–III to generate a test function from a given annotated function, and defines Phase IV to execute the test function with KLEE. At the end of Phase III, *PanicCheck* produces two versions of the generated test function: a human-readable Rust code and an executable version for KLEE. Phase IV feeds KLEE with the generated executable version to reveal panics. Given an annotated function, *PanicCheck* executes all phases by issuing the command "`cargo-verify --backend=KLEE --tests`". This command performs two tasks: (T1) to build the Rust program as well as all available test functions using the Rust compiler, and (T2) to execute the built code with KLEE. We implemented Phases I–III as an integral macro, which rewrites Rust code by creating and adding in a parametrized test. The macro is then loaded in the compilation process (T1), where it receives the token stream of annotated function from compiler for syntax-tree creation, analysis, and manipulation. Phase IV corresponds to the execution process (T2). Because we did not do anything in particular for Phase IV, we will focus our discussion on Phases I–III.

3.1 Context Extraction

Given the token stream of an annotated Rust function, Phase I uses a parsing library—`syn` [19]—to parse tokens. It also invokes APIs of `syn` to traverse the resulting syntax tree in order to locate the function signature, which includes the function's name, parameter list, and parameters' data types.

3.2 Symbolic Variable Creation

For each parameter extracted in Phase I, Phase II determines how to create a corresponding symbolic variable processable by KLEE. So far, *PanicCheck* can provide full or partial support for the symbolic variable creation of 28 data types. These 28 types include 18 primitive types, 4 compound types, and 6 collection types.

3.2.1 Primitive Types. As shown in Table 1, by calling the RVT APIs `data_type::abstract_value()`, *PanicCheck* fully supports variable generation for 16 of the 22 primitive types. It also fully supports the unit type. Because the unit type has only one value "`()`", we do not need to generate any symbolic variable for the data type, neither does KLEE need to enumerate values. *PanicCheck* partially supports variable generation for the reference type. It can declare symbolic variables for shared (i.e., immutable) references, but not for exclusive (i.e., mutable) references. Typically, to generate a reference variable of type T (i.e., &T), *PanicCheck* needs to first create a symbolic variable of type T (e.g., `u8`), and then use the reference to that variable as the created reference variable (e.g., of type `&u8`). Due to the time limit, we did not implement *PanicCheck* to generate syntax trees or code for exclusive references. We plan to address this limitation in the future, by extending our current parser implementation as well as the templates for code generation.

Among the remaining four types, *PanicCheck* does not support `fn` or pointer as KLEE does not handle pointers well. This is because the memory address space is huge; KLEE can easily get stuck with the state explosion issue when symbolizing a pointer to enumerate address values. Notice that our treatments for references and pointers are totally different because in Rust, even though references and pointers have the same underlying data—addresses for some memory, they have different constraints and semantics with the compiler [20]. Namely, references have rules enforced by the compiler: (1) they cannot outlive what they refer to (the "referent"); (2) mutable references cannot be aliased. References behave like the variables they point to. They have a type, and developers can interactive with that type to read it or (with mutable references) modify it. On the other hand, pointers are semantically more about addresses. When developers interact with pointers, they modify addresses instead of the variables pointed to. When they print pointers without using the `unsafe` keyword, addresses are printed out.

Additionally, *PanicCheck* does not support slice or str. Both slice and str are dynamically sized types—types without a statically known size or alignment [21]. Because Rust must know the size and alignment of things in order to correctly work with them, dynamically sized types can only get used via references (e.g., `&str`) and parameters of these types must be declared as references.

3.2.2 Compound Types. *PanicCheck* provides partial tool support for four compound types: array, enum, struct, and tuple. Two reasons can explain why the array type is not fully supported. First, developers can declare arrays to have arbitrary lengths. When an array variable contains a very large number of elements (e.g., >30), *PanicCheck* needs to define many independent symbolic variables, adding them to an array in order to generate a symbolic array variable. When enumerating possible states of all those element symbolic variables, KLEE will encounter the state explosion problem and work ineffectively to reveal panics. Second, when an array has a compound or collection type as its element type, e.g., array of arrays, too many primitive-typed independent variables can be nested into the array level-by-level, making KLEE fail. Based on our experience, KLEE can respond in a timely manner when an array has at most 30 primitive-typed elements, so we built *PanicCheck* accordingly.

PanicCheck does not fully support enum or struct because both types allow developers to define custom data structures. While custom data structures can be very different from each other, the elements of a custom data structure can also have complex data structures. It can be very challenging to properly generate symbolic variables for such data types. Therefore, currently *PanicCheck* only supports variable creation for three widely used built-in types: Option, Result, and String. In the future, we will conduct more advanced static program analysis to characterize custom data structures, and extend *PanicCheck* to generate symbolic variables for those structures.

Rust allows each tuple to have 2–11 elements. However, if a tuple has some compound-typed or collection-typed elements, the total number of independent variables in the tuple can become too large for KLEE to explore. To ensure that KLEE can often respond to *PanicCheck* in a timely manner, we built *PanicCheck* to only model tuples that are declared to have primitive-typed elements.

3.2.3 Collection Types. *PanicCheck* provides partial support for six collection types: Vec, VecDeque, LinkedList, BTreeMap, BTreeSet, and BinaryHeap. This is mainly because each collection can have an arbitrary number of elements. When elements are symbolized as independent variables, there is no way that KLEE can fully support the state enumeration for all variables' value combinations. Consequently, we set the length of Vec, VecDeque, and LinkedList to 30 based on our experimental experience with KLEE. We noticed that KLEE becomes extremely slow and usually produces no output if the length goes beyond 30. We set the length of BTreeMap and BTreeSet to 10. This length is smaller than 30, mainly because the data types leverage B-Tree, a data structure more complex than vectors and lists. We set the length of BinaryHeap to 5 also because of the complexity of the internal data structure.

PanicCheck does not support HashMap or HashSet, because KLEE often wastes time verifying the hashing algorithm used in Rust [22, 23] instead of verifying the actual program logic.

Table 1: *PanicCheck*'s creation of symbolic variables for different Rust data types

Category	Data Type	Tool Support	Details
Primitive Type	bool	Full	Call the RVT's API `bool::abstract_value()` to create a symbolic variable.
	char	Full	Call `u32::abstract_value()` to create a symbolic variable. Then call `char::from_u32(c).unwrap_or_reject()` to ensure that the variable only holds values in [0, 0xD800) or (0xDFFF, 0x10FFFF]—corresponding to valid characters.
	f32, f64	Full	Call `data_type::abstract_value()` to create a symbolic variable, where `data_type` can be f32 or f64.
	fn	No	*PanicCheck* does not generate any symbolic variable for function pointers (fn), as KLEE does not handle pointers well.
	i8, i16, i32, i64, i128, isize	Full	Call `data_type::abstract_value()` to create a symbolic variable, where `data_type` can be i8, i16, i32, i64, i128, or isize.
	pointer	No	*PanicCheck* does not generate any symbolic variable for pointers, as KLEE does not handle pointers well.
	reference	Partial	*PanicCheck* generates symbolic variables for shared (i.e., immutable) references, but does not generate symbolic variables for exclusive (i.e., mutable) references.
	slice	No	There is no pass-in parameter to have the data type slice. Instead, a parameter can have the data type of slice reference, which is fully supported by *PanicCheck*.
	str	No	There is no pass-in parameter to have the data type str. Instead, a parameter can have the type of str reference, which is fully supported by *PanicCheck*.
	u8, u16, u32, u64, u128, usize	Full	Call `data_type::abstract_value()` to create a symbolic variable, where `data_type` can be u8, u16, u32, u64, u128, or usize.
	unit	Full	The unit type has exactly one value "`()`". When a function parameter has the unit type, *PanicCheck* generates the constant value instead of creating any symbolic variable, and sends that value to KLEE.
Compound Type	array	Partial	*PanicCheck* can generate a symbolic array variable for the array type [T;n], where the element type T must be primitive and n is in [1, 30]. To create such a variable, *PanicCheck* first declares an array variable with the size specified. It then repetitively defines symbolic variables of type T and adds those variables to the array. The partial support is delimited by KLEE's capability.
	enum	Partial	*PanicCheck* creates variables for two enum types—Option⟨T⟩ and Result⟨T, E⟩—Rust built-in types widely used to define function parameters. Although developers are allowed to define their own enum data types, self-defined enum data types often have distinct structures. Thus, *PanicCheck* now cannot generate variables for those types.
	struct	Partial	*PanicCheck* generates variables for one built-in struct—String—a built-in data type widely used to define function parameters. Although developers can also define their own struct data types, *PanicCheck* does not support variable generation for those self-defined data types now.
	tuple	Partial	*PanicCheck* generates variables for tuples with 2–11 primitive-typed elements, because Rust allows at most 11 elements in a tuple.
Collection type	Vec	Partial	For Vec⟨T⟩, *PanicCheck* generates a vector of 30 T-typed elements. Each element is a symbolic variable separately generated for primitive-type T, and then added to the vector.
	VecDeque	Partial	For VecDeque⟨T⟩, *PanicCheck* generates a queue of 30 elements, with each element a symbolic variable separately generated for primitive-type T.
	LinkedList	Partial	For LinkedList⟨T⟩, *PanicCheck* creates a list of 30 elements, with each element a symbolic variable of primitive-type T.
	HashMap	No	KLEE does not handle HashMaps well.
	BTreeMap	Partial	For BTreeMap⟨K, V⟩, *PanicCheck* generates a map of 10 entries, where each entry's key and value are separately symbolic variables of primitive-types K and V.
	HashSet	No	KLEE does not handle HashSets well.
	BTreeSet	Partial	For BTreeSet⟨T⟩, *PanicCheck* generates a set of 10 elements, with each element a symbolic variable of primitive-type T.
	BinaryHeap	Partial	For BinaryHeap⟨T⟩, *PanicCheck* a heap with five independent symbolic variables. The type T must be primitive.

3.3 Test Generation

Since function-to-verify (FTV) shares a common pattern, we define a template in *PanicCheck*. The template contains symbolic variable declaration and FTV call. For each FTV, *PanicCheck* generates two semantically equivalent versions of one test function: (1) source code and (2) LLVM bitcode.

3.3.1 Source Code Generation. *PanicCheck* generates tests based on templates. It has a code template predefined for each data type it supports (as listed in Table 1) to declare variables; it also has a predefined template to call FTV with the newly declared symbolic variables. Actually, to simplify code generation and the static reasoning of data types, *PanicCheck* implements the Strategy design pattern [24] in code templates. The pattern allows us to define alternative algorithms for a specific task (i.e., generating variables given a data type), while *PanicCheck* decides the actual algorithms to use at runtime depending on FTV. Fig. 2 illustrates our strategy-based software design for the generated test code. Here, bold text highlights the newly generated Rust trait and implementations, while plain text describes the trait predefined by RVT.

During the test generation for FTV, a trait (analogous to Java interface) named `Strategy1` is always declared; it declares a uniform function interface `value_gen(...)` that is callable by tests to declare symbolic variables. For each parameter type declared by FTV (e.g., `bool`), *PanicCheck* defines an implementation (e.g., `impl Strategy1 for bool`) to implement the declared trait and function; the implemented function invokes API `AbstractValue::abstract_value()` as needed to create symbolic variables. Note that RVT declares and defines the trait `AbstractValue`, so that the type that implements this trait can always generate variables processable by KLEE. Because our software design follows the Strategy design pattern, the test function *PanicCheck* creates is semantically equivalent instead of fully identical to the one shown in Listing 1.

3.3.2 LLVM Bitcode Generation. The `cargo-verify` command issued by *PanicCheck* (see the beginning of Section 3) automatically converts the source code of generated test function into LLVM bitcode. Thanks to the command usage, *PanicCheck* does not need to implement anything to enable the conversion. In this conversion process, the command also (1) injects value checks to guard critical instructions and (2) inserts `panic!` macros for any potential failure of value checks. All such insertions are automated by KLEE.

Because the compiled LLVM bitcode is hard to read and explain, to facilitate presentation, we use Rust code in Listing 3 to present the semantics of compiled LLVM bitcode for `decode_var(...)`. In the code, we use "..." to omit less important code details. As shown in the

Listing 3: A Rust program used to illustrate the program semantics of compiled LLVM bitcode for `decode_var(...)`

```
1   ...
2   for b in src.iter() {
3     let msb_dropped = b & DROP_MSB;
4     // the following if-statement to check for and report
    ↪  shift-operation overflow (lines 5-7) is injected in the
    ↪  compilation process
5     if (shift > 64) {
6       panic!("... panicked at 'attempt to shift operation with
    ↪  overflow', ...", ...);
7     } else {
8       result |= (msb_dropped as u64) << shift;
9       // the following if-statement to check for and report arithmetic
    ↪  overflow (lines 10-12) is also injected in the compilation
    ↪  process
10      if (shift > usize::Max - 7) {
11        panic!("... panicked at 'attempt to add with overflow', ...",
    ↪  ...);
12      } else {
13        shift += 7;
14        ...
15      }
16    }
17  } ...
```

simplified code, the compilation process injects two `if`-statements separately for the bitwise left shift operator (<<) and the plus equals operator (+=). The first `if`-statement ensures that the number of bits specified does not go beyond the total number of bits available in a u64 number (i.e., 64). If `shift > 64`, a panic is generated. Similarly, the second `if`-statement ensures that the result of plus equals does not overflow; if the result is larger than `usize::Max`, the program panics.

With the LLVM bitcode provided, KLEE scans instructions, unrolls `for`-loops as needed, and explores paths reaching `panic`-statements. For each explored path, KLEE synthesizes and solves constraints, to output solutions (i.e., value assignments to symbolic variables) as panic-triggering test inputs. For our motivating example, KLEE detects a panic as described below:

```
running 1 test
  Value src = [128, 128, 128, 128, 128, 128, 128, 128, 128,
128, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]
test test_decode_var ... FAILED

...

thread 'test_decode_var' panicked at 'attempt to shift left
with overflow', src/varint.rs:121:23
```

The output shows the root cause of a panic ("`attempt to shift left with overflow`"), the code location ("`src/varint.rs:121:23`"), and the panic-triggering test input. Such information can help developers debug code and fix any software flaw implied by the panic.

4 EVALUATION

We implemented *PanicCheck* for the Rust version 1.47.0-nightly (2020-08-02). The LLVM release we leveraged is 10.0.0, and the commit of KLEE version we adopted is `c51ffcd377097ee80ec9b0d6f07f8ea583a5aa1d`. To investigate how *PanicCheck* can help with revealing panics in Rust programs, we applied *PanicCheck* to both open-source and closed-source projects. Specifically inside the company ByteDance Ltd., we applied *PanicCheck* to two internal projects (closed-source)[1]. Additionally, we created a dataset of 21 open-source projects by mining crates.io [25]—the Rust community's crate registry, in order to also apply *PanicCheck* to those projects. In the following part of this section, we will first introduce our open-source dataset and the experiment setup. Then we will describe the experiment results.

4.1 The Open-Source Dataset

To build the open-source dataset, we first used the keyword "parser" to search on crates.io for projects with at least 400,000 downloads. We chose this keyword based on the advice of domain experts of formal verification, who mentioned that the Rust libraries related to parsers are less likely to be well tested and thus more likely to suffer from panic issues. Among the popular projects with at least 400,000 downloads, we selected projects using the following criteria:

- The project uses only syntax defined by the Rust 2018 edition, as *PanicCheck* does not support the new grammar features introduced by the Rust 2021 edition.
- The entry functions (see Section 4.2 for definition), i.e., functions we will annotate with `#[panic_check]`, do not use any self-defined (e.g., struct) or complex data types (e.g., HashMap) that are not supported by *PanicCheck*.
- The entry functions do not have any parameter decorated with the lifetime annotation (i.e., the apostrophe character), as *PanicCheck* does not analyze or validate variables declared with the lifetime annotation.

With the criteria mentioned above, we included 20 popular projects into our dataset. Additionally, we noticed that the open-source project GNU core utilities (coreutils) [26] was once used to effectively evaluate KLEE [13]. Thus, we also included a Rust version of coreutils. Please refer to Table 2 for a full list of the open-source projects in our dataset. All these projects have code publicly available at GitHub.

4.2 Experiment Setup

In our evaluation, we did not annotate every single function of subject projects with `#[panic_check]` for two reasons. First, it is very time-consuming to verify every function, although some functions are more important or more frequently executed than the others. Second, as KLEE conducts inter-procedural analysis, it is quite possible that the symbolic execution of some functions can fully cover that of other functions. To verify the most frequently executed functions without incurring too much effort of redundant verification, we decided to annotate only **entry functions** in each selected project. Among all functions within a given project, we chose entry functions using the following criteria:

- If the project has proptests [27] already defined for some functions, we treat those functions as entry functions because developers are likely to apply proptest to the most important functions. Proptest is a property testing framework. It randomly generates inputs to test whether certain properties always hold for a given program; whenever a failure is found, it automatically finds the minimal test case to reproduce the problem.
- If the project has no proptest defined but contains a file lib.rs, we treat all application programming interfaces (APIs) listed in that file as entry functions. The lib.rs file of a project

[1] The experiment was conducted when first four authors were at ByteDance.

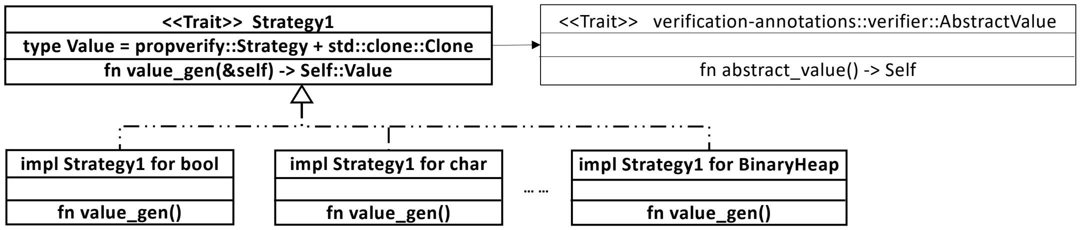
<<Trait>> Strategy1
type Value = propverify::Strategy + std::clone::Clone
fn value_gen(&self) -> Self::Value
<<Trait>> verification-annotations::verifier::AbstractValue
fn abstract_value() -> Self
impl Strategy1 for bool
fn value_gen()
impl Strategy1 for char
fn value_gen()
... ...
impl Strategy1 for BinaryHeap
fn value_gen()

Figure 2: The UML class diagram showing our strategy-based software design for test generation

Table 2: The experiment result on 21 open-source projects

Project	**# of Functions Annotated**	**Build Time without *PanicCheck* (sec)**	**Build Time with *PanicCheck* (sec)**	**Verification Time (sec)**	**Result**	**Details**
https://crates.io/crates/unicode-xid	1	5	111 (22.2x)	26	Pass	The formal verification finishes in 26 seconds.
https://crates.io/crates/unsigned-varint	2	7	135 (19.3x)	3, 4	Pass (2)	The two functions are separately verified with 3 and 4 seconds.
https://crates.io/crates/regex	1	23	118 (5.1x)	>7200	State explosion	KLEE cannot verify the annotated function due to the state explosion issue.
https://crates.io/crates/ryu	2	4	33 (8.3x)	>7200/ function	State explosion (2)	Same as above.
https://crates.io/crates/yaml-rust	1	7	58(8.3x)	>7200	State explosion	Same as above.
https://crates.io/crates/untrusted	1	1	18 (18x)	>7200	State explosion	Same as above.
https://crates.io/crates/csv	1	14	47 (3.4x)	>7200	State explosion	Same as above.
https://crates.io/crates/utf8parse	1	1	20 (20x)	>7200	State explosion	Same as above.
https://crates.io/crates/html5ever	1	8	36 (4.5x)	>7200	State explosion	Same as above.
https://crates.io/crates/der-parser	2	12	27 (2.3x)	>7200/ function	State explosion (2)	Same as above.
https://crates.io/crates/ron	1	14	75 (5.4x)	>7200	State explosion	Same as above.
https://crates.io/crates/url	1	18	42 (2.3x)	>7200	Pass	KLEE can verify some execution paths of the annotated function, but cannot finish verification within 2 hours (the timeout period we specified).
https://crates.io/crates/httparse	1	5	43 (8.6x)	>7200	Pass	Same as above.
https://crates.io/crates/humantime	3	5	34 (6.8x)	>7200/ function	Pass (3)	Same as above.
https://crates.io/crates/nom	1	14	60 (4.3x)	>7200	Pass	Same as above.
https://crates.io/crates/gimli	1	5	94 (18.8x)	>7200	Pass	Same as above.
https://crates.io/crates/lexical	1	14	20 (1.4x)	>7200	Pass	Same as above.
https://crates.io/crates/minimal-lexical	1	2	23 (11.5x)	1	Panic	Attempt to subtract with overflow.
https://crates.io/crates/integer-encoding	1	9	21 (2.3x)	29	Panic	Attempt to shift left with overflow.
https://crates.io/crates/coreutils	100	8–21/ subproject	16–59 (1.8x–6.2x)/ subproject	>7200/ function (44), 11–886/ function (56)	Pass (44) + Clap Panics (45) + Other Panics(11)	The majority of panics are by Claps. When excluding Claps, the panics are about calling unwrap() functions on invalid values, missing function argument, incorrect char boundary, or unexpected invalid UTF-8 code point.

implies the project to be a software library, while the APIs listed in that file are accessible by library users. Thus, those APIs are important to verify.

- If the project has no proptest or lib.rs defined but contains a main function, we treat the function as an entry function. We believe that the main function typically executes the most important functionalities.

With the criteria mentioned above, we annotated in total 125 entry functions in open-source projects and more than 40 functions in closed-source projects. The column **# of Functions Annotated** in Table 2 shows the distribution of the 125 entry functions. In particular, there are 100 subprojects in coreutils, and each subproject defines a main function. Thus, we annotated 100 functions in coreutils. As we conducted all experiments in May 2021–October 2021, all program versions we experimented with were downloaded during that period.

4.3 Experiment Results

In Table 2, the column **Build Time without *PanicCheck*** shows the time cost of purely building each project without involving any step of *PanicCheck*. **Build Time with *PanicCheck*** describes the total time cost of (1) a clean build and (2) the first three steps of *PanicCheck*. Namely, any time difference between the two columns shows the runtime overhead incurred by *PanicCheck*'s first three steps. By comparing the measured values for these columns, we found *PanicCheck* to incur 6–128 seconds to the build procedure.

Namely, *PanicCheck* expanded the compilation overhead by 0.4–21.2 times. Such overheads were introduced macro expansion, *PanicCheck* compilation, and bitcode generation. Thanks to the Rust conditional compilation, such overheads will not affect the build process in production mode because *PanicCheck* is only executed in the testing mode. Thus, developers do not need to remove those macros when building the production binary.

Column **Verification Time** shows the runtime overhead of KLEE execution, corresponding to the fourth step of *PanicCheck*. For 56 subprojects of coreutils and 5 other projects, KLEE execution finished quickly and spent 1–886 seconds on each project. *PanicCheck* either reported no panic after exploring all paths or revealed the first panic it encountered. However, for another 44 subprojects of coreutils and 16 other projects, KLEE execution could not finish within the allocated time—2 hours. In particular, for nine projects, the verification procedure was stuck with the problem of state explosion: there were too many states for KLEE to enumerate. KLEE could not enumerate all states or verify any function. For the remaining (sub) projects, KLEE could not finish its exploration within two hours although it was not stuck with state explosion; its exploration got slowed down by the value enumeration for variables of complex/compound data types or String. We still considered these projects to partially pass formal verification due to a time limit.

Finding 1: *Among the 125 functions annotated for 21 open-source projects, PanicCheck revealed 59 panics for 59 functions but failed to verify 11 functions due to state explosion; 3 functions passed complete verification and 52 functions passed partial verification due to the time limit.*

We also annotated more than 40 functions in 2 closed-source projects of ByteDance. These two projects belong to the Key Management System (KMS). KMS is an internal key management service that other internal services leverage to perform encryption and decryption. The two projects used in our experiment contain several thousand lines of code in total (no more than 10 thousand LOC). They are real-world crucial Rust projects, instead of toy examples crafted by the paper authors for research purposes. Internally, ByteDance requires KMS to have no unrecoverable error, as panics in this service can lead to serious consequences like data loss or service disruption. In our experiment, we applied *PanicCheck* to functions related to certificate parsing, encryption, and decryption, in order to check whether those functions have unrecoverable errors. *PanicCheck* revealed in total two panics in the projects, both of which were later confirmed and fixed by ByteDance developers.

In total, we found 61 panics in 23 projects (21 open-source + 2 closed-source), when we performed the experiment in 2021. To investigate developers' responses to those panics or software bugs, we further examined the more recent version of these programs as of September 2023 (before submitting this paper). to see whether those bugs were already fixed. If a program's latest version could take in the panic-triggering input and execute smoothly, we concluded that developers recently fixed the bug relevant to that panic. Otherwise, we filed a bug issue for each revealed but unresolved panic and sought developers' feedback. So far, we have observed that 52 panics were already resolved by developers before we filed any issue report. We filed 9 reports for the remaining panics; for 7 panics, developers have confirmed the reported issues and fixed bugs accordingly; developers did not respond for 2 panics. The high fixing rate (i.e., 59/61) indicates that KLEE detects crucially important bugs; the high issue-confirmation rate (i.e., 7/9) implies that KLEE's output is quite helpful for developers to understand and fix bugs.

Listing 4: An example to call `unwrap()` function on the value returned by `options.value_of(...)` [29]

```
let duration: Duration = uucore::parse_time::from_str(
↪  options.value_of(options::DURATION).unwrap()). unwrap();
```

Table 3: The root cause of state explosion projects

Projects	Root Cause
ryu	Raw Pointer Operation
yaml-rust	String Enumeration
untrusted	String Enumeration
csv	String Enumeration
utf8parse	String Enumeration
html5ever	String Enumeration
der-parser	String Enumeration
ton	String Enumeration

Finding 2: *PanicCheck revealed 61 panics in 23 projects. So far, 59 of the panics have been addressed by developers. This observation indicates the great quality of PanicCheck's outputs and high relevance of revealed panics.*

We further inspected the content of 61 panics, and recognized two major root causes. First, 45 of the panics share the same error message "unexpected invalid UTF-8 code point". These panics all occurred in subprojects of coreutils, due to the usage of a library clap [28]. When these subprojects passed invalid UTF-8 strings (e.g., `./expand "Ê"`) to a clap API, the API does not properly handle the invalid inputs and thus triggers panics. Recently we observed that the clap developers improved their API implementation, to cause no panic in any of the projects invoking that API.

Second, nine panics are about calling `unwrap()` functions on invalid values. All these panics occurred in subprojects of coreutils. For instance, in the `timeout` subproject of coreutils, `unwrap()` was once called on the return-value of `options.value_of(...)` (see Listing 4). Although developers assumed that `options.value_of(...)` always returns normal values, it turned out that the method call can return an `Err`-typed value. Calling `unwrap()` on that value can trigger a panic and halt the program execution. Developers fixed such bugs by conducting value checks before calling `unwrap()` functions.

Finding 3: *54 of the 61 panics occurred because unexpected or invalid values were used to call method APIs.*

To pinpoint the root causes of state explosion, we annotated functions called by the entry point during execution. However, functions with unsupported features (e.g., lifetime scope annotations) were excluded from annotating. The identified root causes are presented in Table 3. Notably, seven projects faced issues due to string enumeration; given the vast search space of strings or bytes, KLEE could not verify these projects within the specified time. For *ryu*, the parser's pursuit of optimal performance led to the conversion of unsigned integers into raw pointers. This caused

KLEE to enumerate the memory space, significantly expanding the state space and consequently triggering state explosion.

Finding 4: *PanicCheck cannot handle the string and raw pointers cases well due to the limited capability of KLEE.*

5 THREATS TO VALIDITY

Threats to External Validity. All the empirical observations we made so far are based on our experimental dataset. These observations may not generalize well to other Rust programs. In the future, we would like to include more projects into our evaluation, so that our findings can become more representative.

Threats to Construct Validity. Our tool implementation is limited by the Rust edition (i.e., 2018) *PanicCheck* currently targets and the KLEE/LLVM versions it uses. Namely, *PanicCheck* does not support new features introduced by the more recent releases of Rust, neither does it support features that are not well supported by KLEE or LLVM. This is mainly because *PanicCheck* is based on RVT, and RVT targets Rust 2018. Currently, *PanicCheck* does not analyze or validate variables declared with the lifetime annotation, neither does it generate symbolic variables for exclusive (i.e., mutable) references. When running *PanicCheck* on the newer version of Rust code, it will throw an unknown error without producing any unsound result. In the future, we plan to modernize RVT, and extend the modernized version with *PanicCheck*'s implementation for both old and new language features.

6 LESSONS LEARNED

By enabling large-scale usage, *PanicCheck* reveals both strengths and weaknesses of KLEE when it is applied to Rust programs.

6.1 Advantages of Applying KLEE to Rust Programs

Our study confirms that KLEE can generate meaningful test inputs, and reveal unrecoverable errors existing in Rust programs. All errors reported by KLEE are true positives; there is no false alarm (false error) reported by KLEE. Furthermore, it can even effectively identify the unrecoverable errors overlooked by developers or manually developed test suites. One possible reason to explain this phenomenon is that developers may not be good at thoroughly testing Rust programs. When program logic is complex, developers may only focus on the main paths that are frequently executed and majorly check for design errors. Because KLEE systematically explores feasible paths in programs, it is able to capture edged cases. Additionally, KLEE examines software for errors relevant to memory accesses, division/modulo by zero, over shifts, and assertion violations; thus, the errors it finds can complement the design errors that developers focus on.

6.2 Limitations of Applying KLEE to Rust Programs

We noticed that KLEE is inapplicable to generate test cases for many functions in our dataset. Three major reasons can explain KLEE's limited applicability. First, it cannot analyze concurrent programs. Second, it cannot symbolize the size of memory allocation. Third, it provides very limited support for pointers (i.e., memory addresses).

Additionally, even though KLEE is applicable to verify some functions, it cannot finish verification within a reasonable period of time (e.g., two hours) for two reasons. First, it supports a very limited set of built-in collection types (e.g., BinaryHeap). In particular, when a collection variable contains lots of element variables, symbolizing each element variable can make the overall program state space overwhelmingly large, considerably prolonging the verification procedure. Second, KLEE does not support String variables to contain characters from big vocabularies (e.g., ASCII, UTF-8). This is because when a variable can have strings composed of very diverse characters, generating strings of certain format is almost infeasible or computationally expensive.

To better verify Rust programs with KLEE, we plan to improve *PanicCheck* in two ways. First, we will statically analyze programs to learn how developers' customized data types are formulated with primitive data types. In this way, *PanicCheck* can automatically generate symbolic variables for more compound types. Second, when a function-to-verify is called, some of the parameters it takes may require specialized values satisfying certain requirements (e.g., syntax or regular expressions), which values can be very hard for KLEE to generate even though they are not part of the path conditions to trigger panics. We plan to extend *PanicCheck* so that developers can provide concrete inputs for those variables, to accelerate KLEE's exploration process and reveal more panics.

7 RELATED WORK

Our research is related to empirical studies with KLEE and Rust verification tools.

7.1 Empirical Studies with KLEE

People conducted several empirical studies using KLEE [30–38]. Specifically, Wang et al. [31] compared KLEE-based test suites with manually developed test suites. They observed that KLEE-based test suites have advantages in exploring error-handling code and exhausting options, but are less effective on generating valid string inputs and exploring meaningful program behaviors. Such complementarity between KLEE-based tests and human-crafted tests was also observed by Kurian et al. [38], who applied KLEE to generate test cases for safety-critical embedded software.

As a DSE engine, KLEE provides 10 path search approaches. Two of the approaches belong to random search, while eight approaches belong to heuristic search. To investigate which approach performs best, Zhang et al. [37] applied the 10 approaches to 53 GNU coreutil applications. They found that without constraint optimization, one approach of random search (i.e., random path) outperforms the others in terms of the number of completed paths, statement coverage, and branch coverage. Dong et al. [32] did a similar study through analyzing the 33 optimization flags implemented by LLVM but used by KLEE. They observed that on average, applying optimizations makes symbolic execution worse for coreutils applications.

Two studies were conducted to compare alternative implementations of KLEE and its extension [33, 34]. In particular, Kapus et al. [34] compared an implementation of KLEE using a partial solver based on the theory of integers, with the standard KLEE implementation using a solver based on the theory of bit vectors. They did not observe significant differences between the two. Liew et al. [33] compared two alternative implementations of a KLEE extension component: floating-point symbolic execution. They observed that the tools complement each other, and neither offers a silver bullet.

Kim et al. [30] applied symbolic execution tools (CREST-BV and KLEE) and a static analyzer (Coverity) to the same program, to compare their results. The researchers detected six bugs through symbolic execution, none of which were detected by Coverity. Busse et al. [36] hypothesized that if a static analyzer (Clang Static Analysis or Infer) produces (1) a partial program trace, and (2) conditions to trigger a bug, then KLEE can (a) guide its search to prioritize paths following that trace, and (b) prune paths using those conditions. Their experience of implementing the technique highlights two negative results. First, the partial traces are not that useful in guiding search. Second, static analyzers can rarely find non-trivial bugs. Xu et al. [35] developed a dataset of logic bombs and a framework for benchmarking symbolic execution tools automatically.

Our research is different from all the studies mentioned above, as it applies KLEE to verify Rust instead of C code.

7.2 Verification Tools for Rust Programs

Various techniques were recently created to verify Rust programs [6, 7, 39–44]. CBMC [45] is a bounded model checker for C and C++ programs. CRUST [40] and Kani [42] verify Rust programs by translating code into C-like languages and using CBMC. Facebook's experimental MIRAI [43] is an abstract interpreter for the Rust compiler's mid-level intermediate representation (MIR). It explicitly prioritizes a low false-positive rate for bugs rather than a low false-negative rate, and thus does not claim to provide sound verification [42]. Similar to KLEE, Crux-MIR [44] conducts symbolic execution to verify programs written in C/C++ and Rust. However, it models memory usage differently from KLEE.

Prusti [41] is a Rust compiler plugin built on the Viper verification infrastructure [46]. It analyzes information from the Rust compiler and synthesizes a corresponding core proof for the program. To verify correctness properties beyond memory safety, users can annotate Rust programs with specifications at the abstraction level of Rust expressions; the technique waives all annotations into the core proof to verify modularly whether these specifications hold. SMACK [39] is a software verification toolchain that translates LLVM IR code into Boogie intermediate verification language [47], which is verified by Boogie verifiers like Corral [48]. SMACK was initially designed to support Clang as a frontend; Baranowski et al. [49] extended SMACK to also verify Rust code.

Lindner et al. [6, 7] recently proposed two alternative approaches to verify Rust programs based on the KLEE symbolic execution. One approach is contract-based verification [6]. The researchers demonstrated that by properly implementing contracts (i.e., pre- and post- conditions of Rust functions) in Rust programs, they enabled KLEE to find contradictions between contracts, and thus to explore the composite behaviors of functions with reduced complexity. The other approach is annotation-based verification [7]. The researchers demonstrated the new approach using a safety function (eq) from the PLCopen library. Given the function, the researchers first formulated assertions directly from the overall safety properties of the PLCopen specification; then they verified the overall safety with KLEE.

The techniques mentioned above were proposed to verify Rust programs in various ways. Our work is wrapped in the usage of KLEE in *PanicCheck* to explore KLEE's effectiveness in verifying Rust programs. With our wrapping logic, the KLEE could be simply changed to the verification engines mentioned above. Developers can implement the underlying trait to support different verification tools they desired.

8 CONCLUSION

In this paper, we developed *PanicCheck* that relieves the programmers' burden of writing test cases and enables large-scale study of KLEE on Rust programs. We then use *PanicCheck* to carry out a case study to investigate how effectively KLEE can help developers reveal panics in practice. The major findings of our study include: (1) among the functions we studied, KLEE revealed in total 61 panics that reside in 6 projects; (2) 59 of the 61 panics have been addressed by developers; (3) 54 of the panics occurred because unexpected or invalid values were provided to method APIs; (4) KLEE does not work effectively when FTV involves concurrency or complex data types. In the future, we plan to further improve *PanicCheck* to support more Rust-specific features (e.g., lifetime annotation) and to integrate more formal verification techniques (e.g., SeaHorn [50]). In this way, we can assess the verification effectiveness of more techniques, and recommend techniques to developers accordingly.

9 DATA AVAILABILITY

Our data is available at https://github.com/NEUZhangy/ICSE-SEIP-2024.

ACKNOWLEDGEMENT

We thank all reviewers for their valuable feedback. This work was partially funded by NSF CCF-1845446 and NSF-1929701. The first author started working on the project when doing an internship at ByteDance Ltd.

REFERENCES

[1] "Why is Rust programming language so popular?" https://codilime.com/blog/why-is-rust-programming-language-so-popular/, 2021.

[2] "Error Handling," https://doc.rust-lang.org/book/ch09-00-error-handling.html, 2022.

[3] "Rust - Error Handling," https://www.tutorialspoint.com/rust/rust_error_handling.htm, 2022.

[4] "Rust Error Handling In Practice," https://medium.com/coinmonks/rust-error-handling-in-practice-376d86ba12ca, 2023.

[5] "Panics - Comprehensive Rust," https://google.github.io/comprehensive-rust/error-handling/panics.html, 2023.

[6] M. Lindner, J. Aparicius, and P. Lindgren, "No panic! verification of rust programs by symbolic execution," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 2018, pp. 108–114.

[7] M. Lindner, N. Fitinghoff, J. Eriksson, and P. Lindgren, "Verification of safety functions implemented in rust - a symbolic execution based approach," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1. IEEE, 2019, pp. 432–439.

[8] "project-oak/rust-verification-tools," https://github.com/project-oak/rust-verification-tools//, 2022.

[9] A. Reid, L. Church, S. Flur, S. de Haas, M. Johnson, and B. Laurie, "Towards making formal methods normal: meeting developers where they are," *CoRR*, vol. abs/2010.16345, 2020. [Online]. Available: https://arxiv.org/abs/2010.16345

[10] N. Tillmann and W. Schulte, "Parameterized unit tests," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, pp. 253–262. [Online]. Available: https://doi.org/10.1145/1081706.1081749

[11] "Rust/KLEE status update," https://project-oak.github.io/rust-verification-tools/2021/03/29/klee-status.html, 2021.

[12] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, "Push-Button verification of file systems via crash refinement," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

[13] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI'08. USA: USENIX Association, 2008, pp. 209–224.

[14] "Automatic Rust verification tools (2021)," https://alastairreid.github.io/automatic-rust-verification-tools-2021/, 2021.

[15] "Type of bugs that KLEE can find," http://mailman.ic.ac.uk/pipermail/klee-dev/2020-April/001983.html, 2020.

[16] "Crates," https://doc.rust-lang.org/rust-by-example/crates.html.

[17] "integer-encoding 1.1.7," https://crates.io/crates/integer-encoding/1.1.7, 2021.

[18] "Primitive Type slice," https://doc.rust-lang.org/std/primitive.slice.html.

[19] "syn," https://docs.rs/syn/latest/syn/, 2022.

[20] "What's the difference between references and pointers in Rust?" https://ntietz.com/blog/rust-references-vs-pointers/, 2023.

[21] "Basic-Topic-String-and-string-Slice," https://users.rust-lang.org/t/basic-topic-string-and-string-slice/41479, 2020.

[22] "HashMap in std::collections - Rust," https://doc.rust-lang.org/std/collections/struct.HashMap.html, 2022.

[23] "HashSet in std::collections - Rust," https://doc.rust-lang.org/std/collections/hash_set/struct.HashSet.html, 2022.

[24] "Strategy - Rust Design Patterns," https://rust-unofficial.github.io/patterns/patterns/behavioural/strategy.html, 2022.

[25] "crates.io: Rust Package Registry," https://crates.io, 2022.

[26] "Coreutils - GNU core utilities," https://www.gnu.org/software/coreutils/, 2022.

[27] "Proptest Book," https://altsysrq.github.io/proptest-book/intro.html, 2022.

[28] "clap – Rust," https://docs.rs/clap/latest/clap/, 2022.

[29] ""timeout" needs better error message," https://github.com/uutils/coreutils/issues/3040, 2022.

[30] Y. Kim, M. Kim, Y. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 1143–1152.

[31] X. Wang, L. Zhang, and P. Tanofsky, "Experience report: How is dynamic symbolic execution different from manual testing? a study on klee," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 199–210. [Online]. Available: https://doi.org/10.1145/2771783.2771818

[32] S. Dong, O. Olivo, L. Zhang, and S. Khurshid, "Studying the influence of standard compiler optimizations on symbolic execution," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 205–215.

[33] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle, "Floating-point symbolic execution: A case study in n-version programming," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. IEEE Press, 2017, pp. 601–612.

[34] T. Kapus, M. Nowack, and C. Cadar, "Constraints in dynamic symbolic execution: Bitvectors or integers?" in *Tests and Proofs: 13th International Conference, TAP 2019, Held as Part of the Third World Congress on Formal Methods 2019, Porto, Portugal, October 9–11, 2019, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2019, pp. 41–54. [Online]. Available: https://doi.org/10.1007/978-3-030-31157-5_3

[35] H. Xu, Z. Zhao, Y. Zhou, and M. R. Lyu, "Benchmarking the capability of symbolic execution tools with logic bombs," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1243–1256, 2020.

[36] F. Busse, P. Gharat, C. Cadar, and A. F. Donaldson, "Combining static analysis error traces with dynamic symbolic execution (experience paper)," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 568–579. [Online]. Available: https://doi.org/10.1145/3533767.3534384

[37] Z. Zhang, Z. Wang, F. Yang, J. Wei, Y. Zhou, and Z. Huang, "Random or heuristic? an empirical study on path search strategies for test generation in klee," *Journal of Systems and Software*, vol. 188, p. 111269, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000334

[38] E. Kurian, D. Briola, P. Braione, and G. Denaro, "Automatically generating test cases for safety-critical software via symbolic execution," *Journal of Systems and Software*, vol. 199, p. 111629, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223000249

[39] Z. Rakamarić and M. Emmi, "Smack: Decoupling source language details from verifier implementations," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 106–113.

[40] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: a bounded verifier for rust (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 75–80.

[41] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360573

[42] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying dynamic trait objects in rust," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 321–330.

[43] Facebook, "MIRAI," https://github.com/facebookexperimental/MIRAI, 2019.

[44] "Crux-MIR," https://github.com/GaloisInc/crucible/blob/master/crux-mir, 2020.

[45] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176.

[46] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62.

[47] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387.

[48] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 427–443.

[49] M. Baranowski, S. He, and Z. Rakamarić, "Verifying rust programs with smack," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2018, pp. 528–535.

[50] "Seahorn," https://github.com/seahorn/seahorn, 2011.