

Exploring Formal Verification in Rust

Eduardo Sandalo Porto

Who am I?



<https://sandalo.dev>

MSc student @ UFMG

BSc in CS @ USP

Rust was my gateway
into programming
languages & verification

Outline

- What will we be talking about?



What is Formal Verification?

Any ideas?



What is Formal Verification?

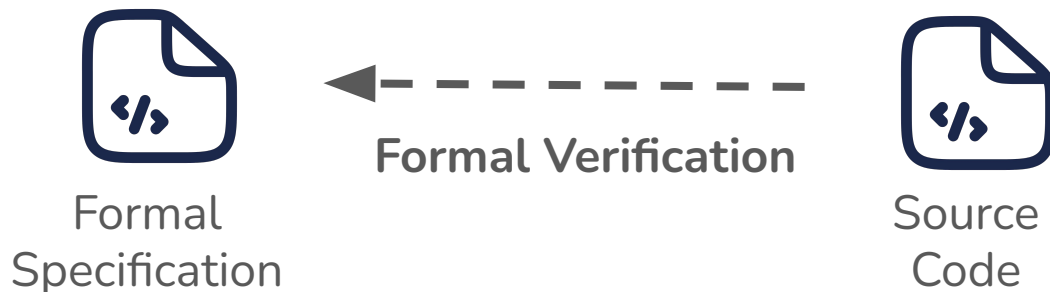
- **Formal Methods**

- Using **formal models** to **reason** (about code)
- Formal models in some **formal** language (i.e., backed in mathematics/logic)
- Some methods include:
 - Specification
 - Synthesis
 - **Verification**

What is Formal Verification?

- **Formal Verification**

- **Proving** that some **code** satisfies some **formal specification**
- Code behavior is described by a **semantics**
- This can be done through:
 - Deductive logical steps
 - State exploration
 - Sound approximations
 - ...
- The model we choose restricts what we can reason about (often, *functional correctness*)



(some) Types of Formal Verification

Deductive Verification (or, Theorem Proving)

- Producing mathematical proofs using deductive reasoning
- Reasoning about the code using logical rules
- “Whenever we have state S and expression E , transit to state S' and expression E' ”

Model Checking

- Check all possible states of a program
- Create finite-state model of the system
- Specify properties you want to be true
- Checker traverses states, prunes invalid transitions

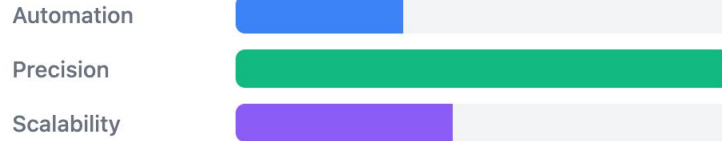
Abstract Interpretation

- Map variables to an imprecise *abstract domain*
- Track some specific detail about these variables over some model (usually a *lattice*)
- Can have *false positives*

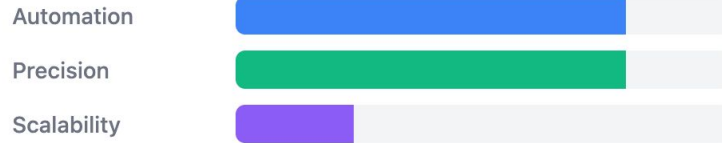


(some) Types of Formal Verification

Deductive Verification (Theorem Proving)



Model Checking



Abstract Interpretation



(very rough comparison)

(some) Types of Formal Verification in Rust

Deductive Verification



Creusot



Prusti



Aeneas

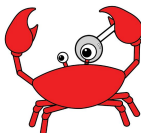


Flux



Verus

Model Checking



Kani



Stateright



ESBMC

Abstract Interpretation

MIRAI



SeaHorn



Rust standard library verification

 Rust Tests passing  Build Book passing

This repository is a fork of the official Rust programming language repository, created solely to verify the Rust standard library. It should not be used as an alternative to the official Rust releases. The repository is tool agnostic and welcomes the addition of new tools. The currently accepted tools are [Flux](#), [GOTO Transcoder \(ESBMC\)](#), [Kani](#), and [VeriFast](#).

The goal is to have a verified [Rust standard library](#) and prove that it is safe.

1. Contributing to the core mechanism of verifying the rust standard library
2. Creating new techniques to perform scalable verification
3. Apply techniques to verify previously unverified parts of the standard library.

For that we are launching a [contest supported by the Rust Foundation](#) that includes a series of challenges that focus on verifying memory safety and a subset of undefined behaviors in the Rust standard library. Each challenge describes the goal, the success criteria, and whether it has a financial award to be awarded upon its successful completion.

<https://github.com/model-checking/verify-rust-std>

Deductive Verification

- From now on, we will talk about **deductive verification**.
- Specifically, we will focus on **Verus** and a subproject **Vest**.



Deductive Verification

(Semi-) automatic verification



Verus



Z3

Z3 SMT Solver

(Semi-) manual verification



Aeneas



LEANTM

Lean Prover

SMT Solvers

- ***Satisfiability modulo theories***

- The problem of determining whether a formula with a given set of theories is **satisfiable**
 - “*There exists a valuation for the variables in the formula that make it true*”
 - Generalization of the SAT problem
 - NP-complete or *harder*
- SMT solvers are programs that attempt to solve an SMT problem
- Very applicable: *AWS calls SMT solvers over a billion times a day [1]*



CVC5 is a state-of-the-art SMT solver whose tech lead is a professor at UFMG!

Floyd-Hoare Logic [5]

- Hoare Triples

- Precondition - Command - Postcondition
- $\{P\} C \{Q\}$

$\{\text{True}\} b := 4 \{b = 4\}$

$\{a = 2\} b := 2 * a \{a = 2 \wedge b = 4\}$

$\{b \geq 5\} b := b + 1 \{b \geq 6\}$

$\{\text{False}\} \text{skip} \{b = 10\}$

$\{\text{True}\} \text{while } i \neq 10 \text{ do } i := i + 1 \{i = 10\}$

obviously, gets much more complex...

$$\frac{}{\{P\} \text{skip} \{P\}} \text{SKIP}$$

$$\frac{}{\{Q[a/x]\} x := a \{Q\}} \text{ASSIGN}$$

$$\frac{\{P\} S \{R\} \quad \{R\} T \{Q\}}{\{P\} S; T \{Q\}} \text{SEQ}$$

$$\frac{\{P \wedge B\} S \{Q\} \quad \{P \wedge \neg B\} T \{Q\}}{\{P\} \text{if } B \text{ then } S \text{ else } T \{Q\}} \text{IF}$$

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \text{while } B \text{ do } S \{P \wedge \neg B\}} \text{WHILE}$$

$$\frac{P' \rightarrow P \quad \{P\} S \{Q\} \quad Q \rightarrow Q'}{\{P'\} S \{Q'\}} \text{CONSEQ}$$



Weakest Precondition Calculus [6]

- Given that I want to end up in a state where Q is true, what is the least restrictive initial state $wp(C, Q)$ required before running C ?
- Thus, if we want Q to be true after running the program, we just need to establish that $wp(C, Q)$ is true before it
- Example:

$$wp(x := E, Q) = Q[x \leftarrow E]$$

$$wp(x := x + 2, x = 5) \equiv (x + 2 = 5) \equiv (x = 3)$$

- Given user supplies preconditions pre , the formula $pre \Rightarrow wp$ is then **negated** and sent to the SMT solver
 - If result is **UNSAT**, the desired property is **true**
 - Gives counter-examples in the SAT case

obviously, gets much more complex...



Weakest Precondition Calculus [6]

- Caveats: *loops*
 - The mechanical algorithm for *wp* extraction could just unroll the loop forever
- We need an **invariant** and a **variant**
 - The *invariant* tells the verifier what we want to be true at each iteration of the loop
 - thus, what we take out from it
 - The *variant* must tend to some final value
 - thus, ensuring termination

obviously, gets much more complex...

Verus [2, 3]



- Automatically applies the *wp* rules across the (annotated) code
 - Then, asks the SMT solver if *the supplied preconditions imply the weakest preconditions*
 - Lots of tools do this for many programming languages, **however:**
- Classic systems-level verification problem: reasoning about *memory*
 - Usually requires complex reasoning using *separation logic*
- **Rust's borrow checker guarantees single-ownership and non-aliasing**
 - The verifier can offload memory safety concerns to the type system
 - Verus can treat uniquely-owned memory as simple mathematical values in the SMT solver
- Additionally, allows the formal specification and correctness proofs of:
 - Normally *unsafe* code in Rust
 - Concurrent code



Verus [2, 3]

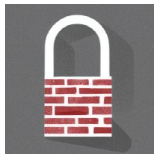


- The basics:
 - **DSL within Rust** as a macro for users to **supply preconditions and postconditions**
 - Three types of code:
 - **spec** (*ghost code*)
 - Defines the desired high-level properties
 - Can only call *spec* code
 - **proof** (*ghost code*)
 - Supplies additional *lemmas, invariant hints, ...*
 - Can call *spec* and *proof* code
 - **exec** (*run-time*)
 - Actual implementation logic
 - Any non-executable code is erased from run-time

Verus [2, 3]

- Example: Fibonacci

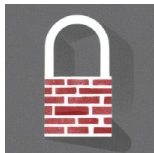
Vest [4]



- A practical, relevant example on using Verus
 - I'm working on a similar project to be presented at SBLP 2026 (CBSOft, São Paulo)
- The problem:
 - Parsing binary formats is hard and a major source of vulnerabilities
 - Format confusion attacks, malleability attacks, side-channel attacks
 - Commonly, hand-written optimizations opens doors for attacks
- The solution:
 - Representing the format in a domain-specific language (DSL)
 - Automatic generation of performant serializers and parsers from that format
 - Automatic proofs of correctness for the generated serializers and parsers



Vest [4]



- The proofs:

Additionally, Vest gives some timing side-channel resistance

Theorem 1 (1st Round Trip Theorem). *For every binary format $\mathbf{F} = (\mathbb{T}, \mathbf{P}, \mathbf{S})$, if $\mathbf{S}(v)$ returns $b \in \mathbb{B}$, then $\mathbf{P}(b) \neq \perp$ and returns $(v, |b|)$, where $|b|$ is the length of b .*

Intuitively, [Theorem 1](#) says that if serializing a structured value gives a byte sequence, then parsing that byte sequence must succeed and return the original value. This defines correctness of the parser with respect to the serializer.

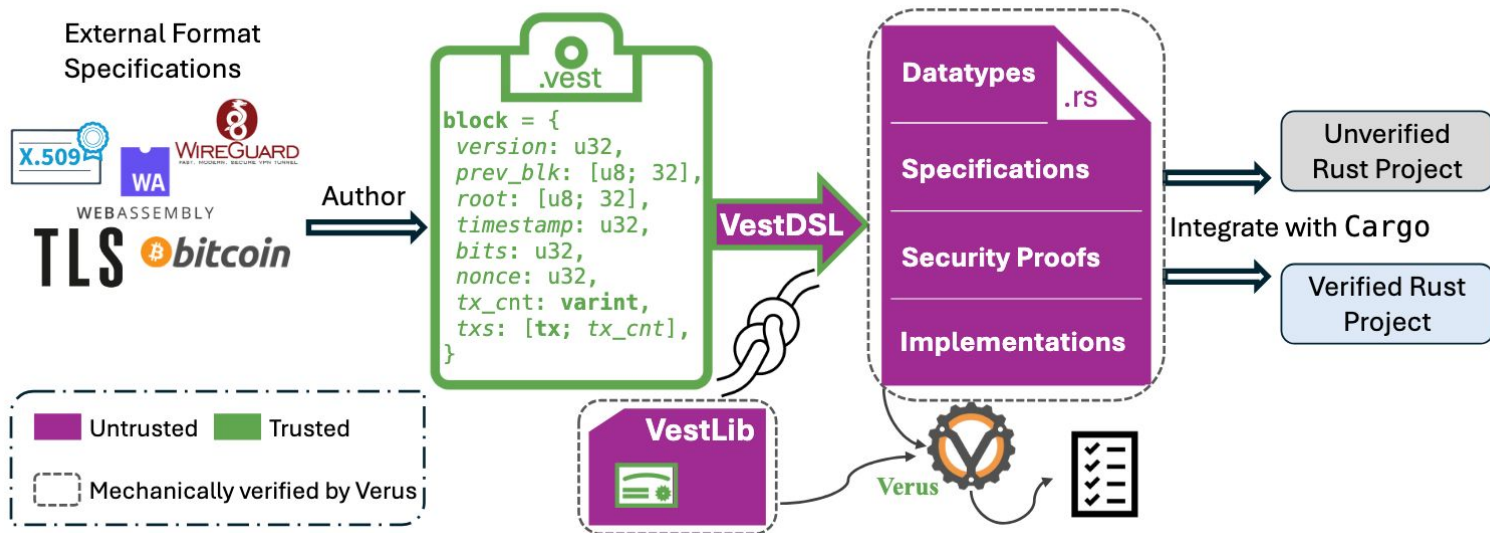
$$\text{Parse}(\text{Serialize}(v)) = (v, \text{length})$$

Theorem 2 (2nd Round Trip Theorem). *For every binary format $\mathbf{F} = (\mathbb{T}, \mathbf{P}, \mathbf{S})$, if $\mathbf{P}(b) \neq \perp$ and returns (v, n) , then $\mathbf{S}(v) \neq \perp$ and returns $b[0, \dots, n]$.*

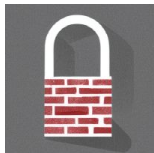
Similarly, [Theorem 2](#) states that if parsing succeeds, then serializing the result must give back the same byte buffer (up to leftover bytes from parsing). This defines correctness of the serializer with respect to the parser.

$$\text{Serialize}(\text{Parse}(b)) = b$$

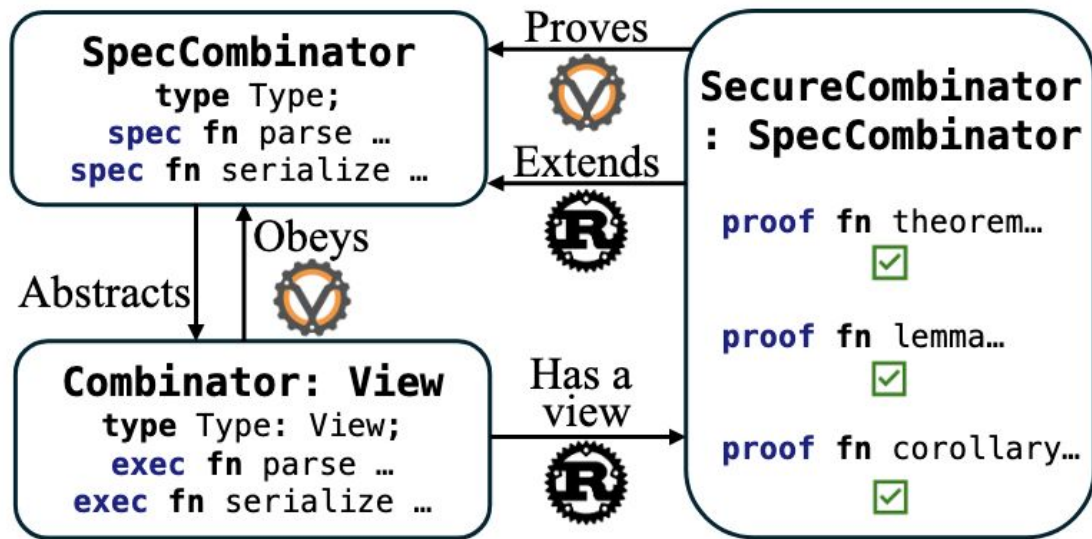
Vest [4]



Vest [4]



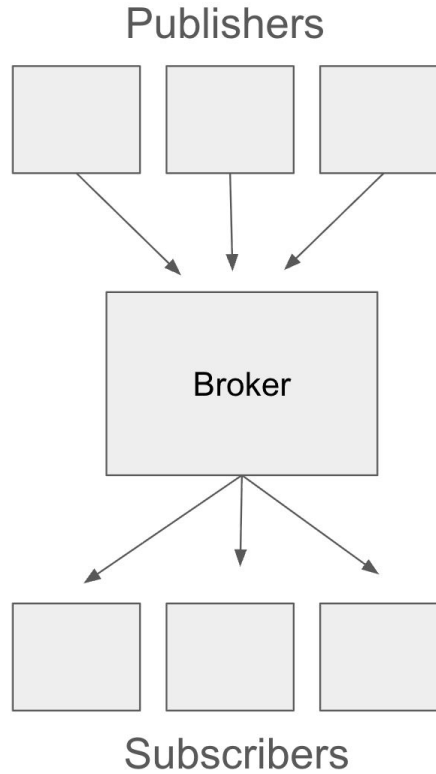
- Three trait system



- A layered library of parser combinators implements each of these traits
- Compositions of these combinators leads to automatic proofs for any generated parser
- (excerpt from the paper)**

Vest [4]

- Example:
MQTT



Vest [4]

- Example: MQTT

Figure 2-2 Fixed Header format

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2...	Remaining Length							

Table 2-3 MQTT Control Packets that contain a Packet Identifier

MQTT Control Packet	Packet Identifier field
CONNECT	NO
CONNACK	NO
PUBLISH	YES (If QoS > 0)
PUBACK	YES
PUBREC	YES
PUBREL	YES
PUBCOMP	YES
SUBSCRIBE	YES
SUBACK	YES
UNSUBSCRIBE	YES
UNSUBACK	YES
PINGREQ	NO
PINGRESP	NO
DISCONNECT	NO
AUTH	NO

Table 2-4 - Properties

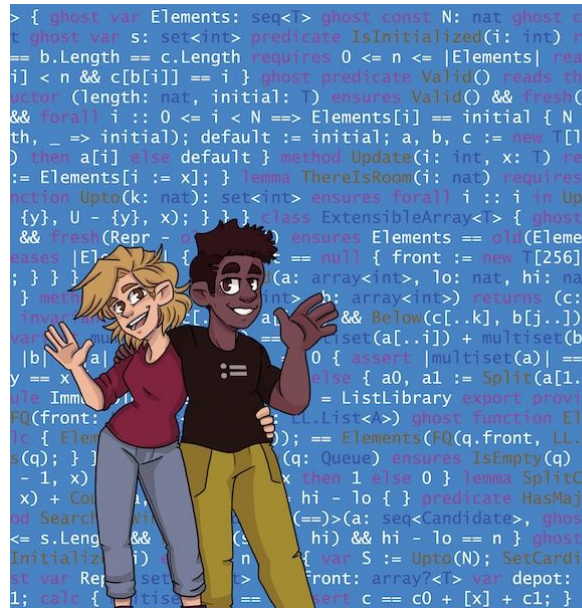
Identifier		Name (usage)	Type	Packet / Will Properties
Dec	Hex			
1	0x01	Payload Format Indicator	Byte	PUBLISH, Will Properties
2	0x02	Message Expiry Interval	Four Byte Integer	PUBLISH, Will Properties
3	0x03	Content Type	UTF-8 Encoded String	PUBLISH, Will Properties
8	0x08	Response Topic	UTF-8 Encoded String	PUBLISH, Will Properties
9	0x09	Correlation Data	Binary Data	PUBLISH, Will Properties
11	0x0B	Subscription Identifier	Variable Byte Integer	PUBLISH, SUBSCRIBE
17	0x11	Session Expiry Interval	Four Byte Integer	CONNECT, CONNACK, DISCONNECT
18	0x12	Assigned Client Identifier	UTF-8 Encoded String	CONNACK
19	0x13	Server Keep Alive	Two Byte Integer	CONNACK
21	0x15	Authentication Method	UTF-8 Encoded String	CONNECT, CONNACK, AUTH
22	0x16	Authentication Data	Binary Data	CONNECT, CONNACK, AUTH
23	0x17	Request Information Problem	Byte	CONNECT
24	0x18	Will Delay Interval	Four Byte Integer	Will Properties
25	0x19	Request Response Information	Byte	CONNECT
26	0x1A	Response Information	UTF-8 Encoded String	CONNACK
28	0x1C	Server Reference	UTF-8 Encoded String	CONNACK, DISCONNECT
31	0x1F	Reason String	UTF-8 Encoded String	CONNACK, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBACK, UNSUBACK, DISCONNECT, AUTH
33	0x21	Receive Maximum	Two Byte Integer	CONNECT, CONNACK

Table 2-5 - MQTT Control Packets that contain a Payload

MQTT Control Packet	Payload
CONNECT	Required
CONNACK	None
PUBLISH	Optional
PUBACK	None
PUBREC	None
PUBREL	None
PUBCOMP	None
SUBSCRIBE	Required
SUBACK	Required
UNSUBSCRIBE	Required
UNSUBACK	Required
PINGREQ	None
PINGRESP	None
DISCONNECT	None
AUTH	None

Takeaways

- Proving programs is hard, but can be very fun
- For many critical systems, a necessity
- Developers can benefit from formal verification without having to prove anything themselves
- May be the key to ensure correctness of LLM-generated code
- LLMs are getting better at writing proofs and specifications themselves



K. Rustan M. Leir
illustrated by Kaleb Leino

PROGRAM PROOFS

Anne Baanen
Alexander Bentkamp
Jasmin Blanchette
Xavier G  n  reux
Johannes H  lzl
Jannis Limperg

The Hitchhiker's Guide to Logical Verification

2025 Desktop Edition
(April 1, 2025)



[lean-forward.github.io/
hitchhikers-guide/2025](https://lean-forward.github.io/hitchhikers-guide/2025)



References

- [1] Rungta, Neha. “A Billion SMT Queries a Day.” Amazon Science, August 18, 2022.
- [2] Lattuada, Andrea, Travis Hance, Chanhee Cho, et al. “Verus: Verifying Rust Programs Using Linear Ghost Types.” *Proceedings of the ACM on Programming Languages* 7, no. OOPSLA1 (2023): 286–315.
- [3] Lattuada, Andrea, Travis Hance, Jay Bosamiya, et al. “Verus: A Practical Foundation for Systems Verification.” *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (New York, NY, USA), SOSP ’24, November 15, 2024, 438–54.
- [4] Cai, Yi, Pratap Singh, Zhengyao Lin, et al. “Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust.” 2025, 6917–35.
- [5] Baanen, Anne, Alexander Bentkamp, Jasmin Blanchette, Johannes Hölzl, and Jannis Limperg. 2025. *The Hitchhiker's Guide to Logical Verification*. Vrije Universiteit Amsterdam.
- [6] Leino, K. Rustan M. *Program Proofs*. MIT Press, 2023.