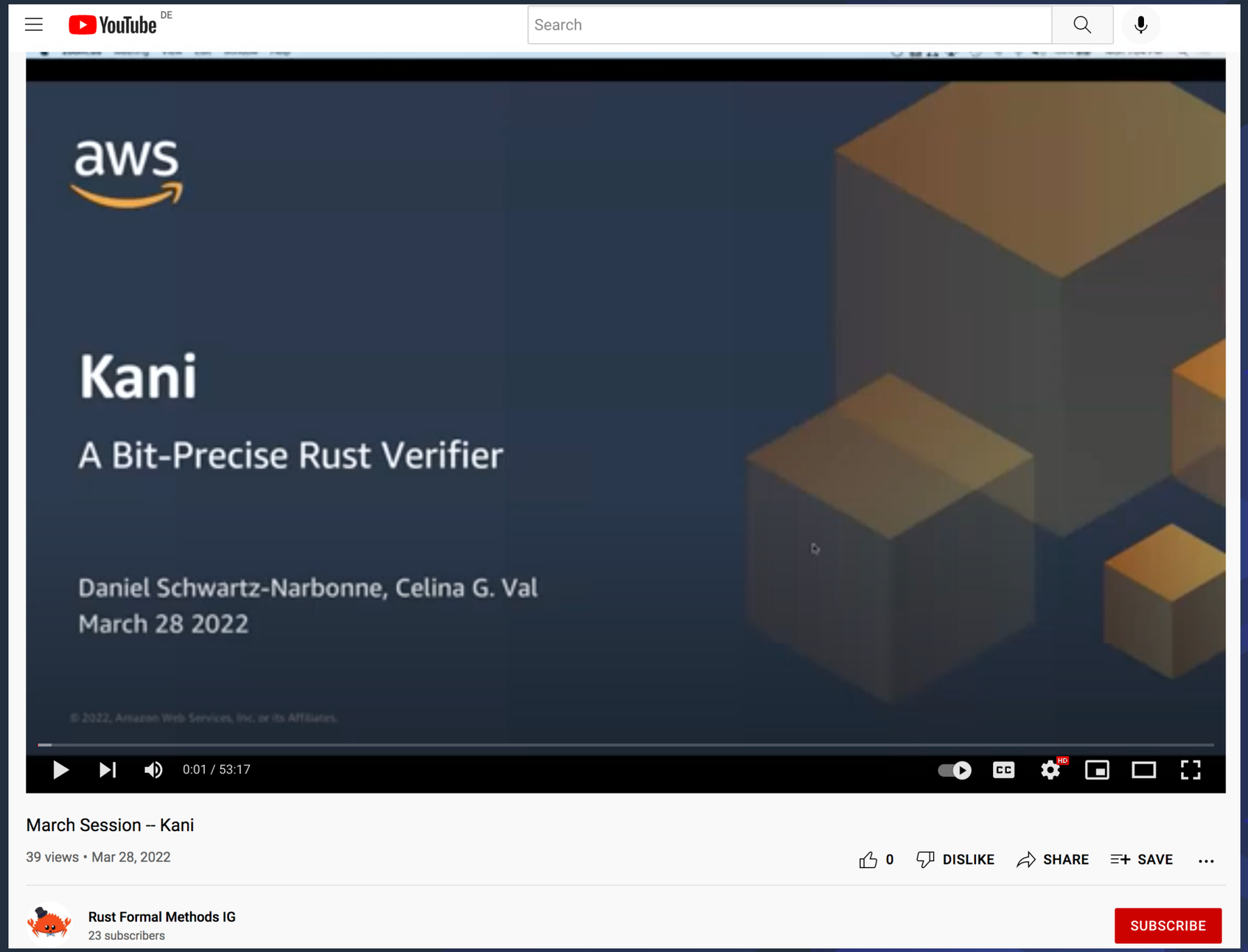
Kani

A Bit-Precise Rust Verifier

Zyad Hassan

May 19, 2022



Demo

Initial users: Systems Programmers

Requirements:

1. Bit precision
2. Supporting unsafe code
3. Heap modeling

Verifying Rust systems code requires bit-precision

```
 96  #[cfg_attr(not(test), rustc_diagnostic_item = "OsStr")]
 97  #[stable(feature = "rust1", since = "1.0.0")]
 98  // FIXME:
 99  // `OsStr::from_inner` current implementation relies
100  // on `OsStr` being layout-compatible with `Slice`.
101  // When attribute privacy is implemented, `OsStr` should be annotated as `#[repr(transparent)]`.
102  // Anyway, `OsStr` representation and layout are considered implementation details, are
103  // not documented and must not be relied upon.
104  pub struct OsStr {
105      inner: Slice,
106  }
```

https://doc.rust-lang.org/stable/src/std/ffi/os_str.rs.html#98

Verifying Rust systems code requires supporting unsafe operations

```rust
25      // The compiler isn't smart enough to remove all of the bounds checks so we resort to
26      // `get_unchecked`.
27      //
28      // https://godbolt.org/z/45cG1v
29
30      // iterate until we reach one of the ends
31      while from_index < from.len() && to_index < to.len() {
32          let from = unsafe {
33              // Safety: this length is already checked in the while condition
34              debug_assert!(from.len() > from_index);
35              from.get_unchecked(from_index)
36          };
37
38          let to = unsafe {
39              // Safety: this length is already checked in the while condition
40              debug_assert!(to.len() > to_index);
41              to.get_unchecked_mut(to_index)
42          };
```

https://github.com/aws/s2n-quic/blob/main/quic/s2n-quic-core/src/slice.rs

Verifying Rust systems code requires a precise model of the heap

```rust
/// Trait implemented by the underlying `MmapRegion`.
pub(crate) trait AsSlice {
    /// Returns a slice corresponding to the data in the underlying `MmapRegion`.
    ///
    /// # Safety
    ///
    /// This is unsafe because of possible aliasing.
    unsafe fn as_slice(&self) -> &[u8];

    /// Returns a mutable slice corresponding to the data in the underlying `MmapRegion`.
    ///
    /// # Safety
    ///
    /// This is unsafe because of possible aliasing. Accesses done through the resulting slice
    /// are not visible to dirty bitmap tracking functionality (when present), and have to be
    /// explicitly accounted for.
    #[allow(clippy::mut_from_ref)]
    unsafe fn as_mut_slice(&self) -> &mut [u8];
}
```

https://github.com/rust-vmm/vm-memory/blob/main/src/mmap.rs

Properties checked

Automatic checks

- Buffer overflows
- Pointer safety
- Division by zero
- Pointer arithmetic
- Arithmetic overflows

User defined properties

- Assertions
- Object invariants

Note that this does not (yet) cover all Rust UB

- Type safety
- Invalid bit patterns
- Aliasing violations

> ⚠ ***Warning:*** The following list is not exhaustive. There is no formal model of Rust's semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. The following list is just what we know for sure is undefined behavior. Please read the Rustonomicon before writing unsafe code.

Formal methods in the development workflow

2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

Code-Level Model Checking in the Software Development Workflow

Nathan Chong
Amazon

Byron Cook
Amazon
UCL

Konstantinos Kallas
University of Pennsylvania

Kareem Khazem
Amazon

Felipe R. Monteiro
Amazon

Daniel Schwartz-Narbonne
Amazon

Serdar Tasiran
Amazon

Michael Tautschnig
Amazon
Queen Mary University of London

Mark R. Tuttle
Amazon

ABSTRACT

This experience report describes a style of applying symbolic model checking developed over the course of four years at Amazon Web Services (AWS). Lessons learned are drawn from proving properties of numerous C-based systems, e.g., custom hypervisors, encryption code, boot loaders, and an IoT operating system. Using our methodology, we find that we can prove the correctness of industrial low-level C-based systems with reasonable effort and predictability. Furthermore, AWS developers are increasingly writing their own formal specifications. All proofs discussed in this paper are publicly available on GitHub.

particular, formal specification of code provides precise, machine-checked documentation for developers and consumers of a code base. They improve code quality by ensuring that the program's *implementation* reflects the developer's *intent*. Unlike testing, which can only validate code against a set of concrete inputs, formal proof can assure that the code is both secure and correct for all possible inputs.

Unfortunately, rapid proof development is difficult in cases where proofs are written by a separate specialized team and *not* the software developers themselves. The developer writing a piece of code has an internal mental model of their code that explains why, and under what conditions, it is correct. However, this model typically remains

https://ieeexplore.ieee.org/document/9276622

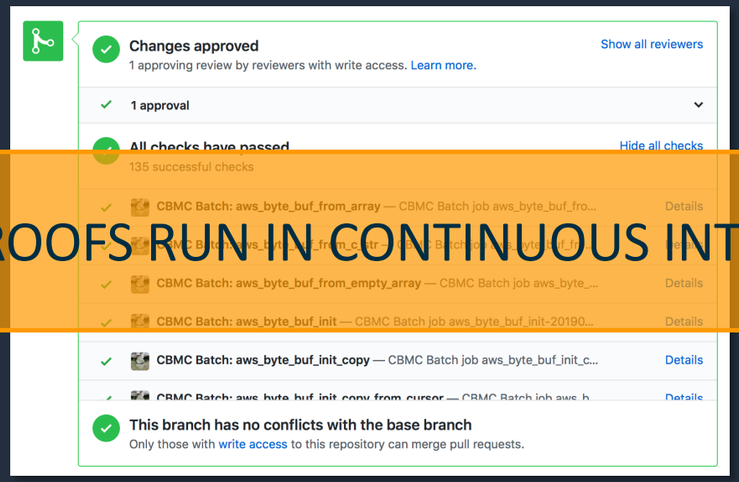
Key insight: use artifacts that fit within developers workflow

```c
/**
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0.
 */

#include <aws/common/array_list.h>
#include <proof_helpers/make_common_data_structures.h>

/**
 * Runtime: 9s
 */
void aws_array_list_pop_front_harness() {
    /* data structure */
    struct aws_array_list list;

    /* assumptions */
    __CPROVER_assume(aws_array_list_is_valid(&list));

    /* save current state of the data structure */
    struct aws_array_list old = list;
    struct store_byte_from_buffer old_byte;
    save_byte_from_array((uint8_t *)list.data, list.current_size, &old_byte);

    /* perform operation under verification and assertions */
    if (!aws_array_list_pop_front(&list)) {
        assert(list.length == old.length - 1);
        assert(list.data);
        assert(list.alloc == old.alloc);
        assert(list.current_size == old.current_size);
        assert(list.item_size == old.item_size);
    } else {
        /* In the case aws_array_list_pop_front is not successful, the list must not change */
        assert_array_list_equivalence(&list, &old, &old_byte);
    }
    assert(aws_array_list_is_valid(&list));
}
```

WRITE SPECS IN THEIR LANGUAGE

```c
AWS_STATIC_IMPL
int aws_array_list_pop_front(struct aws_array_list *AWS_RESTRICT list) {
    AWS_PRECONDITION(aws_array_list_is_valid(list));
    if (aws_array_list_length(list) > 0) {
        aws_array_list_pop_front_n(list, 1);
        AWS_POSTCONDITION(aws_array_list_is_valid(list));
        return AWS_OP_SUCCESS;
    }

    AWS_POSTCONDITION(aws_array_list_is_valid(list));
    return aws_raise_error(AWS_ERROR_LIST_EMPTY);
}
```

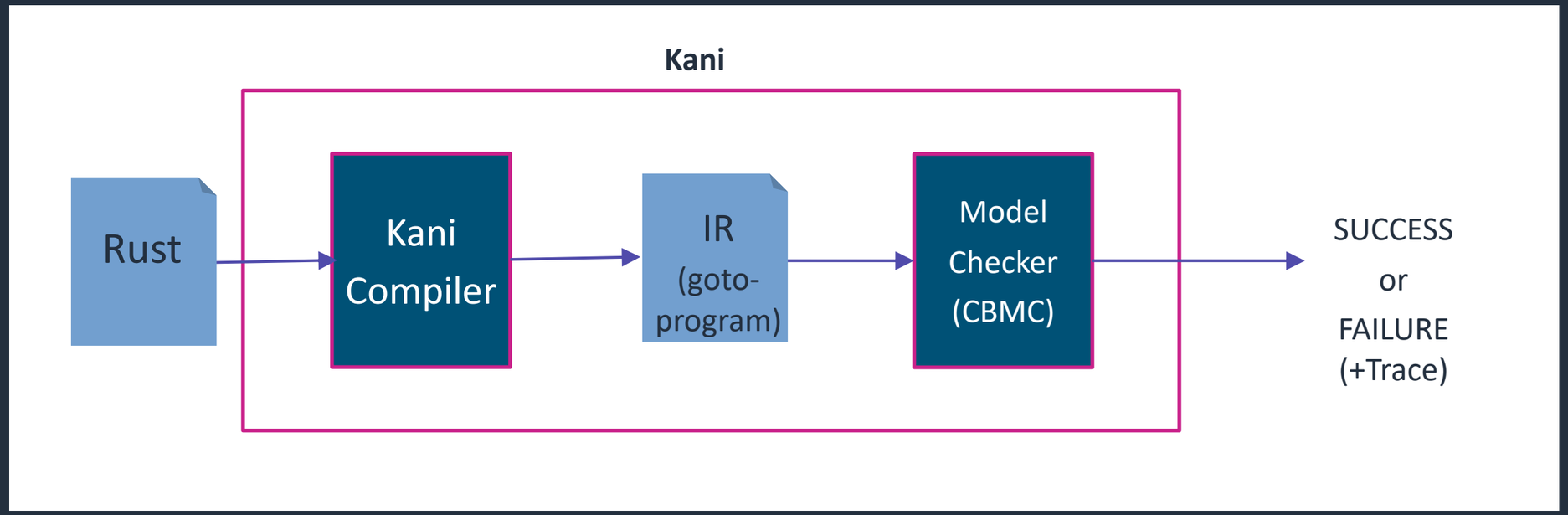
EMBED THE SPECS IN THEIR CODE



PROOFS RUN IN CONTINUOUS INTEGRATION

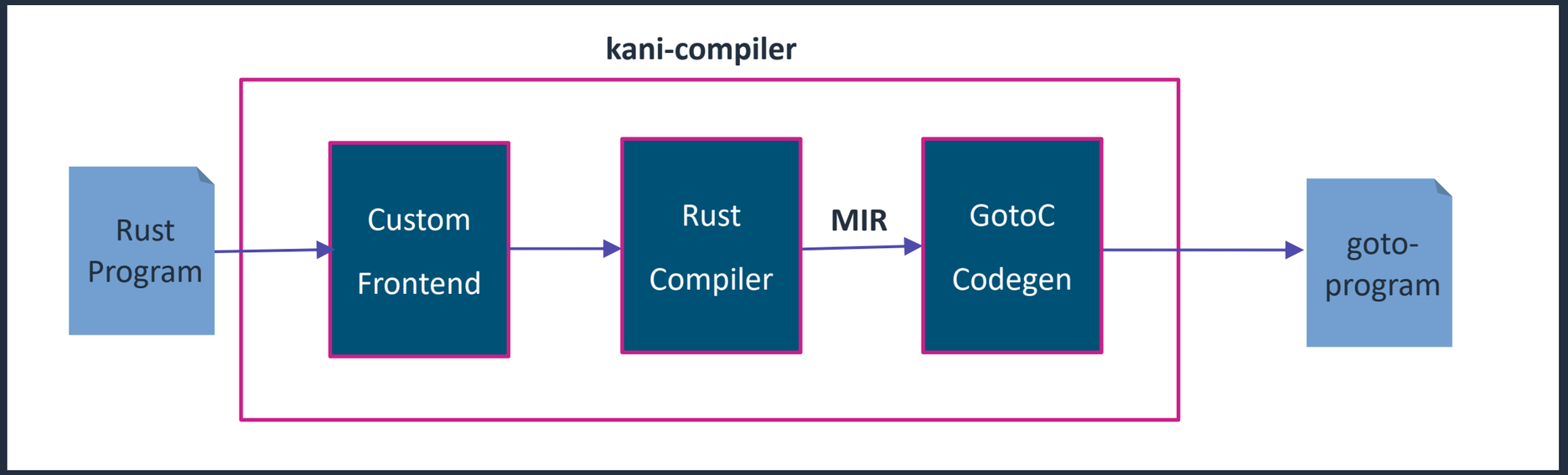
Under the Hood

High Level Flow



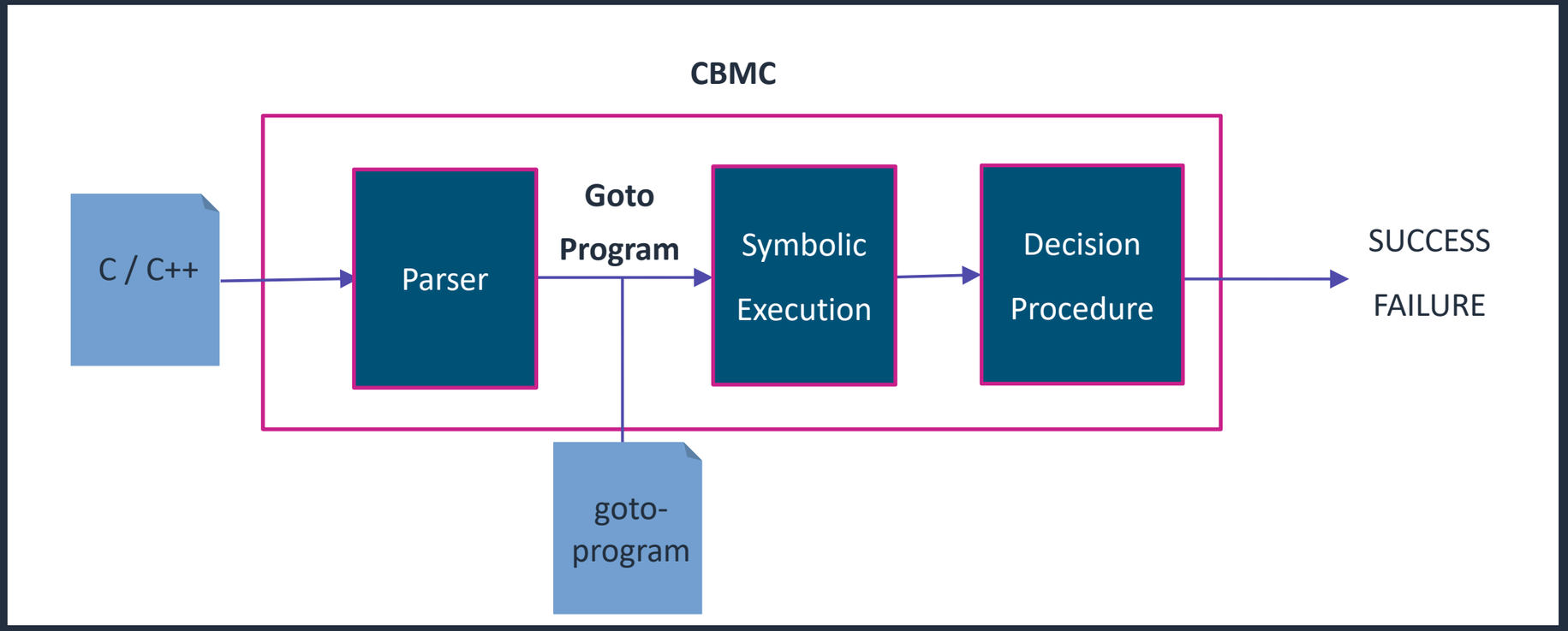
Kani Compiler

- Adds verification checks
- Includes a custom codegen that translates MIR to goto-program
  - MIR: Mid-Level Intermediate Representation
  - **MIR is the source of truth**

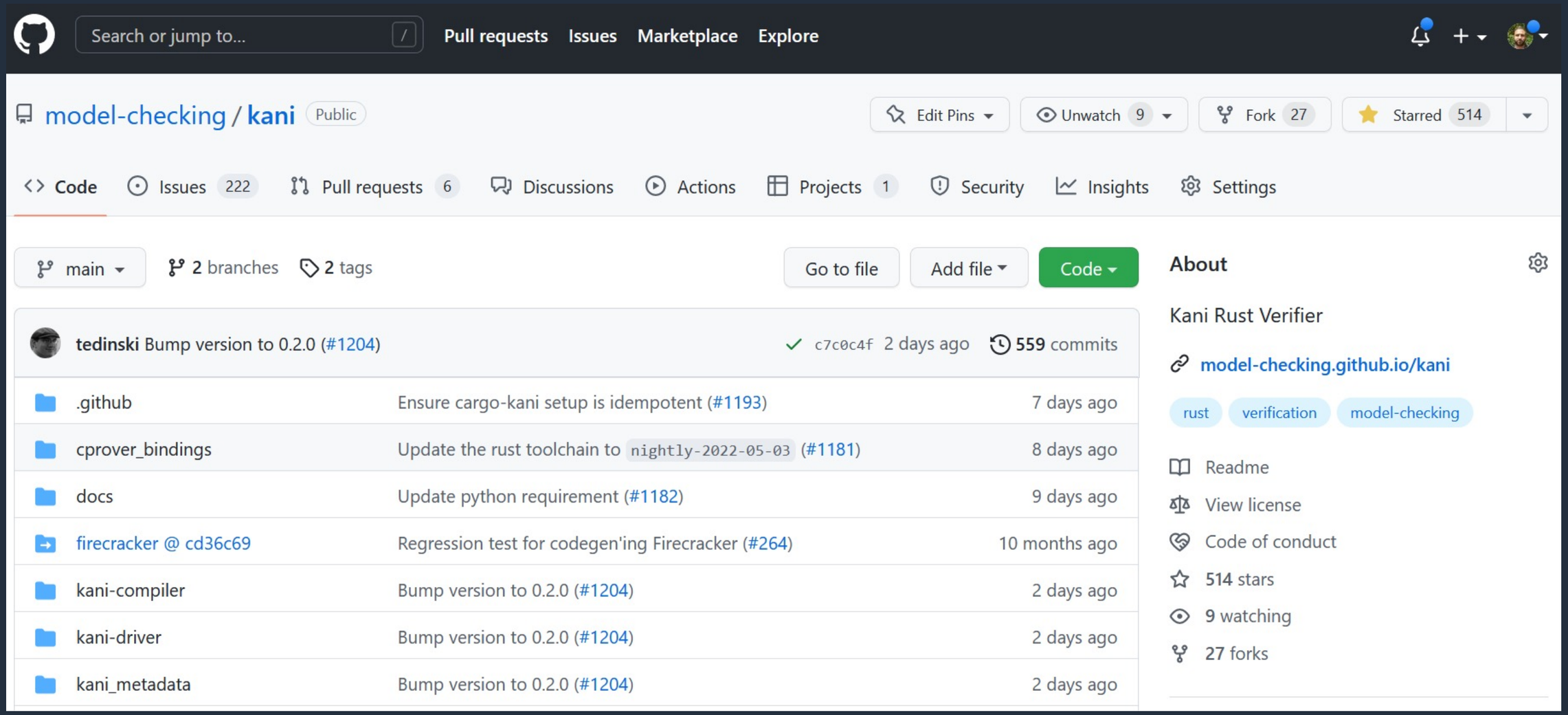


Model Checker: CBMC

- Bounded Model Checker developed for C and C++ programs.
- Verifies memory safety, undefined behavior, user-specified assertions...
- Uses a MiniSat based solver for bit-vector formulas by default.



Open Source



Summary

- Systems programmers need
    - Bit precision
    - Precise modeling of heap
    - Precise modeling of unsafe operations
- Integrate into the developer workflow
    - Unit/Prop test like “proof harnesses”
    - Run using `cargo`
    - Concrete debug traces
- Leverage well established tools
    - Rust Compiler
    - CBMC as bit-precise solver

Appendix

Architecture