The challenge posting for transmute is divided into 4 main parts. Part 1 (which we refer to throughout this document) focuses on verification of the transmute functions themselves (as opposed to functions that depend on transmute). In this document, we provide an overview of the state of completion of part 1.

Note: our contracts/harnesses are written for transmute_unchecked() rather than transmute() currently, so we use “transmute” as shorthand for “transmute_unchecked” in this document.

We next discuss the sources of UB in the Rustonomicon description of transmute; these aren’t exhaustive, and we have a few more, which we thought about, below.

Main sources of UB:

Firstly, the Rustonomicon (https://doc.rust-lang.org/nomicon/transmutes.html) explicitly lists 5 ways transmute can yield UB (note: this is non-exhaustive). We list them here, and we we also comment on how we deal with them, or why we don’t:

- Creating instance of a type with an invalid state:
    - We handle this via our precondition #[requires(ub_checks::can_dereference(&input as *const T as *const U))]
    - Notes for references:
        - Our precondition does not catch references pointing to invalid types -- maybe can_dereference could be extended to catch this?
        - References also cannot be dangling. We do not know of a way to determine if the reference is dangling from a transmute function contract (e.g., if transmuting a dangling pointer to a ref, we can’t write a precondition that checks that the pointer is not dangling). However, perhaps this could be tracked from the moment the pointer is created until the transmute (and would thus be beyond the scope of part 1).
- Transmute has an overloaded return type:
    - Having an overloaded return type is not necessarily UB -- the UB, if there is any, would be caused by misuse of the type ultimately selected by inference. Therefore, this case needn’t be handled directly
- Transmuting an & to &mut:
    - We don’t handle this since the rustc linter already catches cases where we attempt to transmute & refs into &mut refs.
- Transmuting to a reference without explicitly provided lifetime produces unbounded lifetime:
    - This is beyond the scope of part 1 (since it’s perfectly valid for transmute to produce a ref with an unbounded lifetime -- the problem arises depending on when/how that ref is used).
- Transmuting between compound types with different layouts:
    - We partially handle this:
        - Our type validity precondition catches cases where a field in the resulting compound type is invalid

            - Note: we can add harnesses to show that it works on a given compound type (e.g., a struct with two fields, one of which is bool), but can't prove this for all possible compound types due to inability to enumerate them
        - It would also be good to verify that padding bytes are not exposed when transmuting from a struct. However, we do not know of a way to do this currently (e.g., a sound method for identifying padding bytes within a function contract). Note: creating a struct with these bytes is not itself UB, only reading these bytes, so this is likely outside the scope of part 1.
    - It should be noted that a proof of any particular property for compound types (e.g., structs) is seemingly not feasible in Kani, as there is no way to enumerate types

In summary, we handle the first case and we argue that the next three don't need to be handled, leaving only the last one as partially completed (but beyond the scope of part 1).

Special int <-> ptr cases:

Beyond these main sources of UB, there are a few more, which may arise when transmuting pointers (https://doc.rust-lang.org/std/mem/fn.transmute.html), namely:

- ptr -> int:
    - In a const context, this is UB unless the ptr was obtained from an int. We see two ways of dealing with this, neither of which appears to be feasible in part 1:
        - Somehow tracking if a ptr was previously an int (this would be beyond the scope of part 1)
        - Writing a contract clause that forbids transmuting any pointer to an int. This has the downside of refusing int -> ptr -> int transmutes (which are allowed). We also do not know of a way to determine in a clause the actual type of a generic-typed argument.
    - In a non-const context, this is discouraged (as it strips provenance), but is not viewed as UB, so we don't plan on handling this case.
- int -> ptr:
    - The resulting ptr is necessarily invalid, but just creating it this way is not itself UB. We do not currently see a reason for creating a ptr that is necessarily invalid. Thus, we see two main avenues here:
        - There is no valid use-case, so we encode a clause to forbid this transmute (but this requires type-specific reasoning that we do not believe to be possible with our generic transmute wrapper)
        - There may be a valid use case, so we only verify that the resulting ptr is not used (which would be beyond the scope of part 1).

In summary, we believe that the various cases involving transmutes between ints and pointers are either not UB by definition, beyond the scope of part 1, or potentially in the scope of part 1 but not currently possible to verify in a transmute wrapper contract.

Other sources of UB:

Consulting the complete list of UB from the Rustonomicon (https://doc.rust-lang.org/reference/behavior-considered-undefined.html), we believe that all cases of UB relevant to part 1 are covered above.

Conclusion:

We believe that every source of UB relevant to transmute is either:

1. Covered by our preconditions
2. Beyond the scope of part 1
3. Potentially within the scope of part 1, but infeasible to catch with a contract clause

Therefore we believe that part 1 of the transmute verification challenge is effectively complete (pending verification of the currently infeasible-to-catch behaviours).