10 STRATEGIES FOR RESOLVING UNIT PROOFING ERRORS

This section contains the resolution strategies we employed to resolve the different verification errors we encountered. We divide these errors to those related to timeouts, coverage gaps and memory safety violations.

10.1 Resolving Timeout Errors

Verification using CBMC proceeds in three distinct phases: loop unrolling, encoding the unrolled program into a satisfiability formula, and solving the formula. We run CBMC in debug mode (using `–verbosity 9`) and monitor its progress through the log file (`cbmc.xml`). If verification exceeds five minutes, we review the verification log to determine the phase in which the process is stalled.

- *Loop unrolling phase:* Verification may stall during loop unrolling, typically due to recursive calls causing repeated function invocations and excessive loop unrolling. This issue arises because CBMC lacks a command-line option to limit recursion depth.
  Resolution: The logs identify the recursive function. We inspect this function and eliminate the recursion by modifying the program to remove the recursive call.
- *Program encoding phase:* This phase is identified when the `Post-processing` step in the verification log fails to complete within five minutes. In such cases, verification typically results in memory exhaustion. To address this, we proactively investigate potential causes of the extended encoding time. Two common issues are: (i) undefined function pointers, which introduce numerous child functions, and (ii) use of `memmove` with unconstrained pointer and size arguments.
  Resolution: We examine the program for the presence of function pointers and `memmove`. For undefined function pointers, we redefine them to call an empty function. For `memmove`, we temporarily remove the instruction to verify if it is the source of the issue. If confirmed, we terminate the verification, as permanently removing `memmove` may alter the program's semantics.
- *Formula solving phase:* If verification reaches the SAT solving phase within five minutes, we allow it to proceed until it either completes or the solver exhausts available memory.
  Resolution: If memory exhaustion occurs, we analyze the child functions invoked by the target program, identify the most complex function, and replace it with a simplified model that returns an unconstrained value. We then rerun the verification to confirm a reduction in the formula size.

10.2 Resolving Coverage Gaps

After verification completes, some reachable lines of code may remain uncovered. We examine the coverage report generated post-verification to identify these uncovered code blocks. For each block, we determine the cause of the coverage gap and apply an appropriate resolution strategy. The primary causes of coverage gaps were as follows.

- *Configuration-dependent code blocks* These code blocks are depend on specific configurations. Consequently, if the relevant configuration is not enabled, the corresponding code remains uncovered.
  Resolution: To ensure coverage, we identify the required configuration and enable it in either the build file or the project-specific configuration file.
- *Incompletely unrolled loops:* In some cases, code blocks following a loop remain uncovered because CBMC only verifies code that is provably reachable. Such blocks are reachable only if the loop terminates, which may require full unrolling for a fixed number of iterations. Additionally, certain blocks may be conditionally executed based on states reached after a specific number of loop iterations. This also results in a coverage gap even if the immediate post-loop block is covered.
  Resolution: To address this, we identify the loop preceding the uncovered block, analyze the source code, and determine the minimum number of iterations required to satisfy either the loop exit condition or the condition guarding the target block. We then apply a custom unrolling bound for the loop, matching this minimum iteration count.
- *Structs with flexible array members:* C permits structs to include a zero-length or variable-length array as the final member, enabling direct access to data that follows the struct in memory. This pattern is common in protocol implementations, where the struct represents the protocol header and the flexible array accesses the payload. However, in our systematic unit proofs, variables are defined strictly based on their types, and thus a struct or struct pointer is allocated only for the size of the struct itself. As a result, code following the dereference or access of the flexible array member remains uncovered.
  Resolution: When coverage ends at a dereferenced struct pointer, we examine the struct definition to verify the presence of a flexible array member. If present, we modify the corresponding allocation in the unit proof to include sufficient space for the variable-length data following the struct. In cases where the flexible array is cast to another struct, we ensure that the allocated memory for the parent struct is large enough to contain the casted child struct.
- *Statically unreachable code:* Some code blocks remain uncovered because their associated conditions are never satisfied within the context of the unit proof. These instances are identified by analyzing the function's execution flow and determining where the condition governing the block is fixed or constrained, thereby preventing the block from being reached.

10.3 Resolving Memory Safety Violations

When verification reports memory safety violations, our systematic unit proofing method utilizes the error reports, error traces and a set of resolution strategies to identify necessary models of unknown variables that resolve the errors. Unknown variables are those originating outside the target functional unit, including input variables, return values from undefined functions, and global variables. The error traces are used to identify the specific unknown variable that caused the reported violation. The following outlines different error types and the corresponding strategies for deriving variable models to address them.

- *Null Pointer Access Violations:* These violations arise when an unknown pointer variable is accessed without a preceding `NULL` check.

Resolution: We begin by identifying the source of the pointer. If it is an input variable that has already been allocated in the unit proof, we add a non-null pointer precondition (*type one variable model*). If it is a global variable or an unallocated member of an input struct, we modify the unit proof to allocate the variable as a valid pointer and set the corresponding non-null precondition. If the pointer is returned by an undefined function, we construct a model for the function that returns a valid, non-null allocated pointer.

- *Invalid Pointer Access Violations:* These violations occur when a validly allocated pointer is accessed beyond its allocated memory bounds. The violation may result if the pointer or the corresponding access offset is traced to an unknown variable. In some instances, the error arises when the unknown pointer is cast to a struct pointer and subsequently dereferenced.
  Resolution: To resolve such violations, we first identify the source of the pointer and the corresponding invalid offset using the error trace. If the pointer is derived from an unknown variable while the offset is fixed, we model the pointer to reference memory of sufficient size to accommodate the fixed offset (*type three variable model*). If the offset is unknown but the pointer's memory size is fixed, we constrain the unknown offset variable to remain within the bounds of the pointer's memory (*type four variable model*). When both the pointer and offset are unknown, we allow the pointer's memory size to remain unconstrained and model the offset variable to be less than the pointer's allocated size (*type five variable model*). In cases where the unknown pointer is cast to a struct pointer, we ensure the allocated size of the unknown pointer is at least equal to the size of the struct (*type three variable model*).
- *Memcpy/Memcmp/Memmove Read/Write Violations:* These violations occur when the size argument provided to `memcpy`, `memcmp`, or `memmove` exceeds the size of the memory region pointed to by either the source or destination pointer.
  Resolution: We first identify the source or destination pointer and the size argument. Then, following the approach used for *Invalid Pointer Access Violations*, we introduce appropriate variable models for the corresponding unknown variables to ensure memory safety. In cases where the source or destination pointer is defined as an offset from another pointer, we further model the offset and the size variables to ensure that both the offset and the sum of offset and size remain within the bounds of the referenced memory region.
- *Strcpy/Strlen Violations:* These violations occur when an unknown pointer variable is passed as an argument to `strcpy` or `strlen`. Such violations may arise even if the pointer is validly allocated, as these functions continue reading until a null byte is encountered. In the absence of a null terminator, they may read beyond the allocated memory bounds.
  Resolution: We begin by identifying the source of the pointer. If it originates from an unknown variable, we define a new integer variable representing a valid offset within the allocated memory (*type four or type five variable model*). We then assign a null byte to the corresponding offset within the memory region referenced by the pointer. This ensures that a null terminator is present, allowing the function to terminate correctly within bounds.
- *Pointer Relation Violations:* These violations occur when two independent pointers are directly compared. Such violations may also lead to secondary memory safety violations or coverage gaps.
  Resolution: We identify the sources of the involved pointers. If at least, one pointer is traced to an unknown variable, we model that pointer as being at a fixed offset from the other (*type two variable model*), thereby establishing a defined relationship between them.

In some cases, distinct execution flows necessitate different—and potentially conflicting—variable models. To address this, we identify the conditions governing each execution path and associate the corresponding variable models with those conditions (*type five variable model*).