In this chapter, we show that a formally verifiable stack could be constructed in an object-oriented language such as C++ by following certain syntactic restrictions so that C++ methods can be formalized as state-transformer relations as given in Section 4.1. We investigate how to generate an abstract model of the protocol stack, namely stack operations model, in Section 4.2. Section 4.3 looks at issues in construction of verifiable protocol stack. Finally, Section 4.4 describes the proposed split verification approach with an example.

4.1 A State-Transformer View of Internet Protocol Stacks

We develop in this section a formal state-transformer representation for the structure and behaviour of internet protocol stacks.

4.1.1 Primary Operations

Operations on packets performed by protocol stack may be considered as transformations on the state of the protocol stack and the packet/data (which may also be considered to be part of state). We consider protocol stacks having 3 externally visible operations as given in Definition 13.
Definition 13 (Protocol Stack Operations). The three externally visible operations of an internet protocol stack have the following type signatures:

\[
\text{Inbound} : \text{gpdu} \times \text{attrib} \times \text{stack} \rightarrow \text{gpdu} \times \text{attrib} \times \text{stack}
\]
\[
\text{Outbound} : \text{gpdu} \times \text{attrib} \times \text{stack} \rightarrow \text{gpdu} \times \text{attrib} \times \text{stack}
\]
\[
\text{Timer} : \text{attrib} \times \text{stack} \rightarrow \text{gpdu} \times \text{attrib} \times \text{stack}
\]

In the above definition, (a) \(\text{gpdu}\) stands for generic protocol data unit, which represents a sequence of bytes generated or consumed by the protocol stack; (b) \(\text{attrib}\) represents a set of name-value pairs, e.g., the inbound interface name or MAC address for inbound packet or socket address for application data; and (c) \(\text{stack}\) represents the set of primary state-variables of the protocol stack. The \(\text{stack}\) would be internally composed of structures holding state variables for different layers such as IP, ICMP, UDP, or TCP. Each such sub-structure holds its own private set of state variables that are access or modified by operations of the corresponding layer. For example, primary state variables for IP layer would include the list of host interfaces, routing table, packet filtering rules, and packet statistics (for monitoring or diagnostic purposes).

The \(\text{Inbound}\) operation is performed on inbound packets. An inbound packet may get transformed to application data or an outbound packet, or it may get discarded or queued for later processing. The \(\text{Outbound}\) operation is performed on outbound application data. Outbound application data may get transformed to an outbound packet, or may get discarded or queued for later processing. The \(\text{Timer}\) operation is periodically performed to yield an outbound packet or application data which may have been previously queued. It is also possible that an invocation of the \(\text{Timer}\) operation may not yield any output packet or application data. A set of timer attributes is presented as input to these \(\text{Timer}\) transformation along with the state of the stack.

Attributes are typically used for communicating information between layers in the protocol stacks. This avoids the need to directly access or update the state of another layer in the stack. Examples of attributes include the hardware interface MAC address provided by the MAC layer to the IP layer (for \(\text{Inbound}\) operation) or the socket address passed from application to transport layer (for \(\text{Outbound}\) operation). For \(\text{Timer}\) operation attributes may include a timer ID, allowing us to have multiple logical timer operations for different pur-
poses. For example, when a fragmented IP packet arrives, the protocol stack must start a (logical) timer. If rest of the packet does not arrive before the time-out occurs, the queued fragment(s) is (are) discarded. We also need different logical timers for other purposes such as handling TCP time-outs.

We consider threading and interfacing with the data-link layer to be external to the protocol stack. For simplicity we consider that the stack is driven by a single thread.

### 4.1.2 Compositionality of Operations

The protocol stack operations are compositional, meaning that the operations could be constructed following a building-block approach. Compositionality also means that the behaviour of operations is independent of the context and that the meaning of a composite operation is a function of meaning of its parts (sub-operations).

We consider only non-recursive state-transformer operations, though we do admit operations to invoke recursively-defined functions that have no side-effects on the state of the stack (i.e., these function specifications do not contain state variables as free variables, unlike normal state-transformer operations as defined in Section 4.1.3). We found that most of the packet-processing operations can be expressed within these constraints. Exceptions include complex transformations such as those occurring in cryptographic or checksum functions (unless we can express them as side-effect free recursive functions). We may simply abstract them and consider their main properties and introduce these properties into our specification in an axiomatic manner.

Whenever an operation transforms an object to a modified form, we use the term before-state to refer to the object’s state before the transformation and after-state to refer to the object’s state after the transformation. For example, inbound operation transforms before-state of stack to some after-state. As the transformation progresses, the important attributes and properties discovered or accumulated are maintained by `ibuf` and `obuf` types (see below).

**Definition 14** *(Inbound Operation Composition)*. The Inbound operation is composed of three operations having the following types:
In the above definition, (a) the \textit{Init}$_{in}$ operation transforms the input packet, its attributes, and the before-state of the stack to an \textit{ibuf} holding input packet data along with some attributes, an \textit{obuf} holding the output data/attributes, and the after-state of the stack; (b) the \textit{Op}$_{in}$ operation transforms the before-states of input \textit{ibuf}, output \textit{obuf}, and the \textit{stack} to corresponding after-states; and (c) the \textit{Finalize}$_{in}$ operation transforms the input \textit{ibuf}, output \textit{obuf}, and before-state of the \textit{stack} to output data/attributes, and after-state of the \textit{stack}.

During the course of transformation, typical sub-operations of the \textit{Inbound} operation (of type \textit{Op}$_{in}$) would include performing byte-order conversions, inspecting and verifying packet fields, performing de-fragmentation if required, making routing decisions, applying firewall rules, de-multiplexing transport protocol data unit to TCP or UDP modules, and so on. Keeping the input and output of \textit{Op}$_{in}$ to the same type allows us to freely compose sub-operations of the \textit{Inbound} operation. The internal structure of \textit{ibuf} and \textit{obuf} types used in \textit{Op}$_{in}$ may be customized to suit the requirements for a specification stack implementation. Their internal structure is hidden to the outside (e.g., MAC layer or application layer) through the use of \textit{Init}$_{in}$ and \textit{Finalize}$_{in}$ which converts the generic types present in the \textit{Inbound} operation’s signature to and from the private structures. Note that the \textit{stack} type itself would be treated as an opaque data type by operations external to the stack even though it is present in the signature of the stack’s public operations.

The \textit{ibuf} in the input/output parameter lists contains the input IP packet data (possibly transformed) and other related attributes such as offset in the packet being processed by the current layer, input interface for the packet, and essential information extracted from the packet such as its source address and port which may need to be frequently accessed. The \textit{obuf} contains the resultant output data (either IP Packet or application data). In a manner similar to Definition 14, we may define \textit{Outbound} and \textit{Timer} operations as well.

\textbf{Definition 15 (Inbound Operation Instance).} \textit{An instance of inbound operation, inbound::Inbound, may be defined as:}
\[(\text{init}_n::\text{Init}_n) \circ (\text{op}_n::\text{Op}_n) \circ (\text{finalize}_n::\text{Finalize}_n)\]

where, \(\text{op}_n::\text{Op}_n \stackrel{\text{def}}{=} F(\text{op}_1::\text{Op}_n, \text{op}_2::\text{Op}_n, ..., \text{op}_n::\text{Op}_n)\)

In the above definition, \(t::\tau\) represents operation \(t\) having type \(\tau\) and \(\circ\) represents functional composition. We assume that \(f \circ g \circ h(x) \stackrel{\text{def}}{=} h(g(f(x))\). For non-trivial protocol stacks the \(\text{op}_n\) operation may become complex, therefore we may compose \(\text{op}_n\) from smaller transformations having the same type \((\text{Op}_n)\) as shown in the definition. Each of \(\text{op}_j::\text{Op}_n\), for \(1 \leq j \leq n\) may further be composed of transformations of type \(\text{Op}_n\).

### 4.1.3 Primitive and Composite Operations

We follow logic-based specification languages such as Z, and specify an operation using a state transformer formula, which relates values of state variables before the operation (called \(\text{before-state}\)) to values of state variables after the operation (called \(\text{after-state}\)). Names of protocol stack state variables (\(\text{gpdu}, \text{attrib}, \text{or stack}\)) may appear as free variables in state transformer formulas in two ways: (a) as primed names (e.g., \(\text{gpdu}'\)) - these range over possible after-state values of the corresponding state variable, (b) as unprimed names (e.g., \(\text{gpdu}\)) - these range over possible before-state values of the corresponding state variable.

**Definition 16 (Interpretation of State Transformer Formula).** A state transformer formula, \(T\) is interpreted as a function \(F\) on the state space such that the set of pairs in \(F\) satisfy \(T\). The domain of \(F\) represent before-state values and range represents after-state values.

The state space for an operation is determined by its signature. For example, from Definition 14 the state space for \(\text{Op}_n = \text{ibuf} \times \text{obuf} \times \text{stack}\).

We distinguish between composite operations, which are defined in terms of other sub-operations, and primitive operations, which do not depend on any other sub-operations.

**Definition 17 (Primitive Operation).** A primitive operation is specified as: \(p \stackrel{\text{def}}{=} \bigvee_i (c_i \land t_i)\), where \(t_i\) is state-transformer formula relating before-states and after-states of state-variables and \(c_i\) is the before-state condition under which \(t_i\) is applicable (essentially, its pre-condition). We consider only total operations, which means that operation has true as its overall precondition: \(\bigvee_i c_i \iff \text{true}\).
The use of state-transformer formula to specify primitive operations allows us to work with standard first order logic instead of a specialized logic system such as the Hoare logic. This permits easy translation to Z which is based on first order logic and set theory. Considering operations of type $Op_{in}$, we may describe them by specifying the relationship between before-states and after-states of $ibuf$, $obuf$, and $stack$.

**Definition 18 (Composite Operation).** A composite operation can only be constructed by composing one or more primitive or composite operations using: (a) linear composition, (b) guarded composition, or, (c) a combination of linear and guarded composition.

(a) Linear Composition: If $\circ$ represents function composition, then linear composition is defined by: $p \overset{def}{=} p_1 \circ p_2 \circ \ldots \circ p_n$, where, $p_i$ could be either a primitive or composite operation. 

(b) Guarded Composition: Guarded composition is defined by: $p \overset{def}{=} \bigvee_i (g_i \wedge p_i)$, where $g_i$ is called guard formula and $p_i$ is primitive or composite operation.

Guard formulas serve essentially the same purpose as guards in Dijkstra’s guarded command language (GCL) [27].

**Example 1.** An example of a primitive operation is a skip operation of type $Op_{in}$ that leaves the stack unmodified under all conditions, which may be represented as:

$$\text{skip}::Op_{in} \overset{def}{=} ibuf^t = ibuf \land obuf^t = obuf \land stack^t = stack$$

We consider two examples to illustrate guarded composition.

**Example 2.** Consider, an operation $p$ defined as:

$$\lambda ibuf,obuf,stack.\text{if } \text{cond} \text{ then } p_1(\text{ibuf}, \text{obuf}, \text{stack}) \text{ else } p_2(\text{ibuf}, \text{obuf}, \text{stack}).$$

Then, we define its state transformer formula as:

$$p \overset{def}{=} (g_{cond} \wedge p_1) \lor (\neg g_{cond} \wedge p_2)$$

**Example 3.** Consider the following operation which leaves the state unmodified if cond is false: $p = \lambda ibuf,obuf,stack.\text{if } \text{cond} \text{ then } p_1(\text{ibuf}, \text{obuf}, \text{stack}) \text{ else } (\text{ibuf}, \text{obuf}, \text{stack})$, which could be expressed as:

$$p \overset{def}{=} (g_{cond} \wedge p_1) \lor (\neg g_{cond} \wedge \text{skip})$$
4.2 Protocol Stack Operations Model

Although we could directly apply theorem-proving techniques to establish properties of Inbound, Outbound, and Timer operations, this approach quickly becomes unwieldy for stacks of non-trivial size and complexity. Therefore our approach involves applying theorem-proving to establish properties for primitive operations which are then conditionally promoted as properties for the stack specification by performing model-checking on an abstract model of the stack. We call this abstract model the operations model for protocol stacks. The stack operations model is defined as a Kripke structure [39] since we use it to establish promotion conditions which are temporal properties.

The stack operations model captures the sequencing and attributes of packet-processing operations. States represent operation invocations and are labelled with names and attributes of operations. Transitions in the model are decided by the manner in which composite operations are constructed from other primitive or other composite operations. Figure 4.1(a) shows the states and transitions for linear composition of the form given in Definition 18.

![Linear Composition](image)

Figure 4.1: States and transitions for linear and guarded composition in stack operations model

In order to represent guarded composition in the operations model, we introduce an additional guard evaluation operation. The guard evaluation operations are labelled by abstracting the guard formulas using boolean literals that represent the true condition of the guard. If guards are based on propositional variables we may directly use them to label the states of guard evaluation operations.
**Definition 19** (Guard Evaluation Operation). A guarded composition of form given in Definition 18 is transformed to: \( p_0 \circ \bigvee_i (f_i \land p_i) \), where \( p_0 = \bigwedge_i g_i \iff f_i' \). If a guard \( g_i \) also appears in negated form as another guard \( g_k = \neg g_i \), then we may use the same flag variable \( f_i \) in its negated form for \( g_k \).

The states and transitions in the operations model for guarded composition is given in Figure 4.1(b). The operation \( p_0 \) would be typically combined with the preceding operation for establishing operation-level properties regarding the flag variables. In addition to operation names, the states are also labelled with the predicates on variables, \( mod \) and \( acc \), representing modification or access of variables by an operation.

**Example 4.** Consider a composite operation of the form, \( p_{def} = p_1 \circ \ldots \circ p_k \circ \bigvee_i (g_i \land p_i) \). We may rewrite this as: \( p_{def} = p_1 \circ \ldots \circ p_k' \circ \bigvee_i (f_i \land p_i) \), where \( p_k' = p_k \circ p_0 \).

**Definition 20** (Stack Operations Model). The stack operations model is a Kripke structure defined by: (a) set of states, \( S \), determined by the set of operations, (b) set of initial-states comprising states corresponding to the beginning of Inbound, Outbound, and Timer operations, (c) transition relation, \( R \), determined by the manner in which composite operations are structured, (d) set of atomic propositions, \( AP \), comprising set of operation names, module names \(^1\), predicates \( mod \) and \( acc \) on the set of state variables \( V \), and boolean flag variables representing guard evaluation, and (d) labelling function \( L : S \rightarrow 2^{AP} \).

The set of states \( S \) is determined as follows. Primitive operations are represented by a corresponding state in the model. For each composite operation, two states appear in the model, one representing the beginning of the operation and one denoting the end of the operation. In addition, for guarded composition, guard evaluations are represented by states in the model. Figure 4.1 shows the transitions induced by linear and guarded compositions.

**Definition 21** (Labelling Function for Stack Operations Model). Consider state \( s \) corresponding to operation \( p \). Let \( \alpha \) be the set of labels assigned to state \( s \), i.e., \( (s, \alpha) \in L \). If \( p \)

---

\(^1\)A module is simply a logical container for a sequence of associated operations.
is a primitive or composite operation, then the state is labelled by both operation name and
module name, i.e \((p, m) \in \alpha\), if operation \(p\) belongs to module \(m\). If \(p\) is a primitive opera-
tion, then for each variable \(v\) modified by \(p\), the predicate \(\text{mod } v \in \alpha\) and for each variable \(v\)
accessed by \(p\), the predicate \(\text{acc } v \in \alpha\). If \(p\) is a guard evaluation operation inside composite
operation \(q\), then \((q, f) \in \alpha\), where \(f\) is the boolean flag corresponding to the guard.

We analyse the complexity of the operations model with respect to the number of states
and transitions, since this is an important parameter that decides the scalability of our ap-
proach. For the purpose of analysis, we consider protocol stacks having generic compositional
structure as given in Definition 22.

**Definition 22 (Generic Compositional Structure).** Generic compositional structure has three
primary operations at the top of compositional hierarchy. A compositional hierarchy consists
of \(k\) layers of composite operations linearly composed of one or more guarded composite opera-
tions, each having one or more guarded sub-operations. Depending on the primary operation,
the compositional hierarchy may start at layer 1 or layer \(k\). A subset of guarded composite
operations at layer \(1 \leq j < k\) (\(1 < j \leq k\)) are constructed using composite operations of layer
\(j + 1\) (\(j - 1\)). The remaining guarded composite operations at layer \(1 \leq j \leq k\) are constructed
from primitive operations of the same layer. At layer 1 (layer \(k\)) all the guarded composite
operations are constructed from primitive operations of the same layer.

Note that a complex protocol layer may need to be implemented using more than one
compositional layer of Definition 22.

**Theorem 4.2.1 (State Space Complexity for Stack Operations Model).** The worst-case state
space complexity for the stack operations model of a protocol stack having generic composi-
tional structure with: (a) \(k\) layers; (b) each linear composite operation composed of \(n\) guarded
composite operations; and (c) each guarded composite operation composed of \(m\) guarded sub-
operations, is \(O(m^k \times n^k)\).

**Proof.** In order to consider the worst-case state space complexity, we assume all the guarded
composite operations at a layer are composed of composite operations of the next layer. Let
\(N_S(k)\) be the number of states in the stack operations model of a protocol stack with \(k\) layers,
having generic compositional structure. Let \( N_S(k) \) be the number of states introduced by one primary operation. Consider a stack having only one layer \((k = 1)\). In Figure 4.1, if we replace each operation in the linear composition with a guarded composition having \( m \) guards, we get, \( N_S'(k) = 2 \times n + 2 \times m \times n \) and \( N_S(k) = 3 \times N_S'(k) + c \), where \( c \) represents the constant additive factor introduced by \textit{Init} and \textit{Finalize} operations.

If the stack has two layers, then each operation in the guarded composition of first layer generates \( N_S'(1) \) states by composing operations from its next layer. Therefore, \( N_S'(2) = 2 \times n + m \times n \times N_S'(1) \) and \( N_S(2) = 3 \times N_S'(2) + c \). If the stack has \( k+1 \) layers, \( N_S'(k+1) = 2 \times n + m \times n \times N_S'(k) \), \( k \geq 1 \) and \( N_S(k+1) = 3 \times N_S'(k+1) + c \).

Since \( N_S(k) \) and \( N_S'(k) \) differ only by constant factors, they are of the same order.

We need prove that \( N_S(k) = O(m^k \times n^k) \), which we do by induction. \( N_S'(1) = O(m \times n) \), from above. Assume, \( N_S'(k) = O(m^k \times n^k) \) (inductive hypothesis). From above, \( N_S'(k+1) = 2 \times n + m \times n \times O(m^k \times n^k) \), and therefore, \( N_S'(k+1) = O(m^{k+1} \times n^{k+1}) \). Hence, \( N_S'(k) = N_S(k) = O(m^k \times n^k) \).

\[ \textbf{Theorem 4.2.2 (Transitions Complexity for Operations Model).} \textit{The worst-case space complexity for transitions in the stack operations model of a protocol stack having generic compositional structure with: (a) \( k \) layers; (b) each linear composite operation composed of \( n \) guarded composite operations; and (c) each guarded composite operation composed of \( m \) guarded sub-operations, is } O(m^k \times n^k). \]

\textbf{Proof.} In order to consider the worst-case space complexity for transitions, we assume all the guarded composite operations at a layer are composed of composite operations of the next layer. Let \( N_T(k) \) be the number of transitions in the stack operations model of a protocol stack with \( k \) layers, having generic compositional structure. Let \( N_T'(k) \) be the number of states introduced by one primary operation. Consider a stack having only one layer \((k = 1)\). From Figure 4.1, \( N_T'(1) = (3 \times m + 1) \times n \) and \( N_T(1) = 3 \times N_T'(1) + c \), where \( c \) is the constant factor introduced by \textit{Init} and \textit{Finalize} operations.

If the stack has two layers, then \( N_T'(2) = N_T'(1) + m \times n \times N_T'(1) = N_T'(1) \times (m \times n + 1) \). In general, if the stack has \( k+1 \) layers, \( N_T'(k+1) = N_T'(k) \times (m \times n + 1) \) and \( N_T(k+1) = 3 \times N_T'(k) + c \).

Since \( N_T(k) \) and \( N_T'(k) \) differ only by constant factors, they are of same order.
We need to prove that $N_T(k) = O(m^k \times n^k)$. We proceed by induction. $N'_T(1) = O(m \times n)$ from above. Assume, $N'_T(k) = O(m^k \times n^k)$ (inductive hypothesis). From above, $N'_T(k + 1) = N'_T(k) \times (m \times n + 1)$ and therefore $N'_T(k + 1) = O(m^{k+1} \times n^{k+1})$. Hence, $N'_T(k) = N_T(k) = O(m^k \times n^k)$.

4.3 Constructing Verifiable Stacks

In order to construct verifiable protocol stacks, we must decide on the following:

- A language for constructing the stack operations. We use an imperative subset of C++ for this purpose, retaining the class construct but without inheritance.

- A language for constructing proofs. We chose the Z language for this purpose, since its schema construct allows specification of the state-transformer relationship of primitive operations and its schema calculus allows specification of composite operations.

- A strategy for mapping constructs from the C++ subset to Z. Our choice of a specific subset of C++ allows us to formulate methods written in this subset as state-transformer operations.

4.3.1 Operations in Z Notation

We may represent structures holding state variables such as stack or gpdu using state schemas in Z notation and primitive operations using operation schemas. Note that in the Z notation, the $\Delta$ operator denotes state change and $\Xi$ operator denotes that the state remains unchanged.

**Definition 23 (Primitive Operations in Z).** A primitive operation of the form given in Definition 17 is represented in Z notation using an operation schema of the form:

$p \triangleq [\Delta ibuf, \Delta obuf, \Delta stack | (c_1 \land t_1) \lor (c_2 \land t_2) \ldots \lor (c_n \land t_n)]$

**Example 5.** skip :: $Op_m$ may be represented in Z as:

$skip \triangleq [\Xi ibuf, \Xi obuf, \Xi stack]$
Composite operations may be represented in Z using its schema composition and schema calculus constructs.

**Example 6.** Linear composition may be represented by schema composition of the form: 
\[ p \triangleq p_1 \; ; \; p_2 \; ; \; \ldots \; ; \; p_n, \] 
where \( ; \) represents schema composition in Z. Guarded composition may be represented by schema conjunction and disjunction: 
\[ p \triangleq (g_1 \land p_1) \lor (g_2 \land p_2) \ldots \lor (g_n \land p_n), \]
where \( g_i \triangleq [\Xi ibuf; \Xi obuf; \Xi stack \mid guard \ formula_i]. \)

### 4.3.2 Defining the C++ Subset

The C++ subset has the following properties:

- **Primitive and composite operations** are implemented by methods. Primitive methods (rather than any arbitrary statement) are considered to be the fundamental state-transformer units.

- **Global functions** have no side-effects. These may be recursive or may contain iterative loops. These may be translated to axiomatic function definitions in Z.

- **C++ types and type constructors** are mapped to the Z notation. We use only those types and type constructors that permit direct translation to Z types. Table 4.1 shows the C++ types that we use for describing stack operations (see Definition 18) and one possible strategy for mapping C++ types to Z types.

- **Pointer manipulation and heap allocation** are not allowed. Though techniques such as separation logic \[90\] allow reasoning about pointer operations and heap memory objects, we restrict ourselves to a conservative subset of C++ that permits us to construct single-threaded protocol stacks.

Since primitive operations are the fundamental state-transformer units, the after-state value of a variable can be a function of before-state values of a subset of all the state variables, but cannot be dependent on after-state values of any other variable. Of course, when we compose primary operations to form composite operations, we associate after-state values of an operation to the before-state values of the subsequent operation, so this restriction is
<table>
<thead>
<tr>
<th>Abstract type</th>
<th>C++ type</th>
<th>Z type</th>
</tr>
</thead>
<tbody>
<tr>
<td>octet</td>
<td>typedef unsigned char byte;</td>
<td>byte == 0..255</td>
</tr>
<tr>
<td>gpdu</td>
<td>typedef byte[N] gpdu;</td>
<td>gpdu == {s: seq byte</td>
</tr>
<tr>
<td>attrib</td>
<td>enum names = {...}; map(names, long); or struct containing fields for each attribute</td>
<td>names ::= free type definition and define a partial or total injection from names to (\mathbb{Z}) (for numerical attributes)</td>
</tr>
<tr>
<td>ibuf, obuf, stack</td>
<td>class or struct</td>
<td>state schema</td>
</tr>
<tr>
<td>inbound, init, finalize</td>
<td>methods of stack class</td>
<td>operation schema</td>
</tr>
<tr>
<td>(O_{pin})</td>
<td>methods of stack class or other classes whose objects are contained in the stack class</td>
<td>operation schema</td>
</tr>
</tbody>
</table>

Table 4.1: A strategy for mapping of types

Weakened. However, owing to the limitation of the Z language, we do not support recursive composition or recursive definition of primary operations. Since the packet flow in a protocol stack is linear in nature, from lower layers to upper layers or from upper layers to lower layers, the lack of recursive composition does not constrain our specifications. However, recursive or complex functions are still required (e.g., checksum or cryptographic transformations). Currently, we need to introduce these operations as axiomatic function definitions in Z such that the axioms capture either the entire function specifications or important properties of such functions. C++ methods implementing primitive operations may invoke such recursive or complex side-effect-free functions, though they themselves cannot be defined in a recursive manner. Moving to a higher order logic system (such as PVS) will remove these constraints.

C++ methods implementing primitive operations may be represented as Z operation schemas and methods implementing composite operations may be represented in Z using schema composition and schema calculus notations. Our approach would allow automatic translation of the C++ subset that we employ to Z schemas.

**Example 7.** Operations of type \(O_{pin}\) may be constructed in C++ using methods having signature: void op(ibuf\& ib, obuf\& ob) as given below.

```cpp
void op(ibuf& ib, obuf& ob)
{
```
if (ib.flag)
    op2(ib, ob);
    op3(ib, ob);

which may be translated as the following Z schema: \( \text{op} \triangleq (\text{check}_{\text{flag}} \land \text{op}_2) \lor (\neg \text{expr}_1 \land \text{skip}_{\text{in}}) \lor \text{op}_3 \), where, \( \text{check}_{\text{flag}} \triangleq [\Xi_{\text{ibuf}}, \Xi_{\text{obuf}}, \Xi_{\text{stack}}] \text{ib.flag} = \text{TRUE} \)

### 4.4 Split Verification

Given a model \( M \) and a property expressed using formula \( \phi \) (represented using some form of temporal logic), the model-checking problem involves checking whether: \( M \models \phi \) (\( M \) models \( \phi \)). Given a program specification expressed using a set of logic formulae, \( \Gamma \), and a property expressed using formula \( \psi \), the theorem proving method involves checking whether: \( \Gamma \vdash \psi \) (\( \psi \) is provable from \( \Gamma \)). The split verification approach is summarized in Definition 24.

**Definition 24 (Split Verification).** Consider a protocol stack consisting of primary operations Inbound, Outbound, and Timer, built compositionally from a set of primitive operations \( \text{op}_1 \ldots \text{op}_k \) having specifications \( s_1 \ldots s_k \). The stack specification \( S = \bigcup_j S_j \) where \( j \in \{\text{Inbound, Outbound, Timer}\} \) and \( S_j \) represents a primary operation specification. Let \( \text{SOM} \) be the stack operations model corresponding to the stack specification \( S \). Then, given a set of operation-level properties \( p_i \) for primitive operations \( \text{op}_i \) expressed as refinement conditions, and a collection of temporal properties \( \phi_i \) on \( \text{SOM} \), called promotion conditions, split verification checks that \( s_i \vdash p_i \) and \( \text{SOM} \models \phi_i \) for each \( i \).

#### 4.4.1 Operation-level Verification

We may specify operation-level properties in the form of abstract operation schemas that satisfy these properties. These abstract operation schemas may be prepared based on applicable RFCs. At the right level of abstraction it would be easy to show that these abstract operations satisfy the relevant properties. Since these abstract operation schemas are independent of any particular implementation this approach enables us to specify properties
in an implementation-independent manner at the level of RFCs or other standards. As an example, Figure 4.2 shows abstract specification of replay-safe sequence number verification at receiver-side.

The State schema specifies state variables maintained by the replay-safe receiver, which are: (a) received_seq_nums: finite set of sequence numbers; (b) seqnum: the map from input message to sequence number; and (c) body: the map from input message to output message. The set of input messages ReplaySafeMessage and the set of output messages Message are specified as given types. Note that seqnum and body are defined as part of the receiver’s state, which implicitly means that receiver may use some state information (not specified here) for implementing the maps. The ReplaySafeReceive operation checks whether the incoming message’s sequence number is already present in its set of previously received sequence numbers. This is the precondition for the operation (therefore, ReplaySafeReceive is a partial operation). If precondition is satisfied then the state is updated by adding the sequence number to the set of received sequence numbers and providing body of the input message via output variable msg!.

Figure 4.2: Abstract specification for replay-safe receiver-side sequence number verification

We show that a concrete operation of the stack satisfies certain properties by proving that the operation is a correct refinement of an abstract operation that satisfy the same set of properties. For this purpose, we define a retrieve relation mapping the components in the
abstract operation’s state schema to the components in the protocol stack’s state. We need to consider only that portion of the protocol stack’s state which is accessed or modified by the concrete operation under consideration. We then attempt to prove the Z ADT refinement theorems.

4.4.2 Stack Operations Model and Promotion Conditions

In this section, we define and elaborate on the three basic promotion conditions.

Definition 25 (Promotion Conditions). Given a primitive operation \( p \), its promotion conditions are a set of temporal properties in the stack operations model that specifies: (a) Occurrence of \( p \) in relation to other operations, (b) Guard evaluation operations following \( p \), (3) acc and mod predicates on state variables accessed or modified by \( p \).

- Temporal properties regarding occurrence of the operation \( p \) in the Kripke structure in relation to other operations or modules inform us about the influence of \( p \) on the overall packet flow. For example, we may want to establish that for all paths to the tcp module, \( p \) occurs before another operation, say, \( q \).

- As a result of operation-level theorem-proving, we may establish that certain conditions may hold true in the after-state of \( p \). If subsequent operations are guarded by these conditions, then subsequent packet flow is directly influenced by them. Establishing temporal properties regarding guard evaluation for these conditions will provide guarantees on how the subsequent packet flow is influenced by after-state conditions of \( p \). For example, let flag \( f \) represents some condition that holds in the after-state of the operation, then we could establish that guard evaluation operation for \( \neg f \) is never followed by a higher-layer operation, i.e., if condition \( f \) is not ensured by \( p \), then packet is never passed onto higher layers.

- We may want to ensure that values being operated by \( p \) have not been modified by intervening operations or that these values have passed through validation operations before being processed by \( p \). We show this property by establishing temporal properties for acc and mod predicates on variables accessed and modified in the operation. For
example, we could establish that variables modified by $p$ have not been modified by another operation except byte-order conversion. For such verifications, we could have an exception list of operations that perform non-interfering modifications (e.g., byte-order conversion).

### 4.4.3 Split Verification of Echo/UDP/IP Stack

As an example, we consider verifying a stack containing UDP-based Echo service (Echo/UDP/IP). The echo service is a simple diagnostic service based on UDP which responds by echoing request datagrams back to their source IP address and port. The fraggle attack is a well-known denial of service attack exploiting vulnerabilities in UDP Echo service. In fraggle attack, the attacker sends UDP packets to echo service as targeted broadcasts to networks that are known to have a large number of active computers. The source addresses in these packets are spoofed to victim’s address. For each broadcast packet generated by the attacker, all the machines in the intermediary network generate echo response packets to victim’s address, causing bandwidth exhaustion.

Figure 4.3 shows the stack operations model for the Echo/UDP/IP protocol stack, simplified for presentation purpose. Nodes are labelled with operation name and essential attributes of operations such as the layer name and names of packet fields modified by the operation. Since packet flow follows control flow in protocol stacks, the model is an abstract representation of both the packet flow as well as the control flow. Both the forward and return paths of control flow is represented using separate nodes. For example, the node labelled $UDP \land \text{drop}$ represents drop operation at $UDP$ layer. This operation is invoked by deliver operation of $UDP$ layer (denoted by arrow from $UDP \land \text{deliver}$). After the drop operation the packet does not flow up the stack, instead control goes back to the deliver operation (arrow to $UDP \land \text{deliver}$) which may update packet drop statistics.

The security properties that we want to check for our Echo/UDP/IP stack are: (a) The address validation checks as per RFC 1812 (primarily checking for invalid source IP addresses such as private or broadcast addresses) are enforced, and (b) the stack is not vulnerable to fraggle attack. These properties may be further elaborated as:
Figure 4.3: Stack Operations Model for a simplified Echo/UDP/IP protocol stack
1. Operation-level property (to be verified by theorem-proving):
   
   (a) The operation validates source/destination address and source ports as per RFC 1812 and also to thwart fraggle attack. The operation must decide to drop the packet if the validation fails.

2. Promotion conditions based on the stack operations model (to be verified by model-checking):

   (a) The check operation is always performed by IP layer (Definition 25(a)).

   (b) The check operation may result in decision to drop packet (Definition 25(b)).

   (c) If the check operation decides to drop the packet, it is not passed to upper layers (Definition 25(b)).

   (d) No other operation except byte-order adjustment operation modifies the fields verified by the check operation \( src\_addr \) in the example before check is performed (Definition 25(c)).
We represent promotion conditions using branching time logic (CTL) as shown below. We may automatically check whether the operations model (more precisely, the start state of the model) satisfy these properties.

\[
AF(IP \land \text{check}) \quad (4.1)
\]

\[
AF(IP \land \text{check} \implies \text{EXdrop}) \quad (4.2)
\]

\[
AG((IP \land \text{drop}) \implies AG(\neg (UDP \lor TCP))) \quad (4.3)
\]

\[
A((\neg (\text{mod srcip}) \lor ((\text{mod srcip}) \land IP \land \text{ntoh})) \cup (IP \land \text{check})) \quad (4.4)
\]

In the above CTL formula: (a) A and E are path quantifiers, meaning along all paths (inevitably) and there exists a path (possibly), respectively; (b) F, G, U, X means, some future state, globally, until, and neXt state, respectively; (c) IP, UDP, and TCP are module names representing layers in the protocol stack; (d) ntoh and check are primitive operations, the former performing network to host byte-order conversion of packet fields and the latter performing verification of IP packet fields; (e) srcip represents the source IP address field; and (e) drop is the guard evaluation operation.

The operation-level property (Property 1a) may be captured by an abstract Z specification. We could then map the relevant operation in C++ that performs address validity checks to Z notation and verify whether the Z notation obtained from stack code is a valid refinement of the abstract Z specification. This is essentially a proof-by-refinement approach [8].

In order to prove that the concrete stack operation is a correct refinement of the abstract operation, we must first define a retrieve relation, \( R \), that relates the abstract and concrete state schemas.

**Definition 26 (Proof-by-Refinement on Protocol Stack Operations).** Proof-by-refinement on protocol stack operations transformed to Z notation may be done by proving the three refinement theorems for Z schemas [75]:

1. If CI and AI are concrete and abstract initialization schema, then: \( \forall CI \cdot \exists AI \cdot R. \)
2. The concrete operation can be invoked whenever abstract operation can: $\forall R \mid \text{pre } AO \cdot$
pre CO.

3. When abstract operation can be invoked, it will give a result consistent with the concrete
operation: $\forall R; CO \mid \text{pre } AO \cdot \exists A' \cdot AO \land R'$

Definition 26 shows the three theorems that must be proved to show that a concrete Z
specification is a correct refinement of an abstract Z specification. In the definition $A$
and $C$ are concrete and abstract state schemas; $R$ is the retrieve relation between both the
schemas; $AI$ and $CI$ are abstract and concrete initialization schemas; and $AO$ and $CO$ are
abstract and concrete operation schemas. The first theorem shows all initialized elements
in the concrete schema have a corresponding initialized element in the abstract schema as
per the retrieve relation. The second theorem shows that whenever precondition for abstract
operation is satisfied, the precondition for corresponding concrete operation is also satisfied.
The last theorem shows that concrete operation’s results are always consistent with that of
abstract operation (as per the retrieve relation).

Figure 4.4 shows the abstract Z specification of the property to be verified and Figure
4.5 shows the concrete Z specification corresponding to the check operation. The PACKET
type in the figure represents GPDU. IP packet fields are accessed using functions from the
PACKET to the field type (e.g. IPADDRESS). The set of invalid IP addresses $martians$, is
invariantly defined in the $AIP$ state specification. The abstract check operation $A\text{check}$ is
a partial operation which sets the after-state of action to DROP if source or destination IP
address is in the set of martian or broadcast IP addresses. The concrete stack (Figure 4.5)
uses state variables $myaddr$, $mymask$, and $mybaddr$ for holding the IP address, netmask, and
broadcast address for the stack’s network interface. Rather than predefining set of invalid
(martian) addresses, the concrete check operation verifies the subfields of IP address.

Figure 4.6 shows the retrieve relation and the refinement theorems for the IP address
verification (check) operation. The retrieve relation describes relationship between state
variables of abstract and concrete stacks. For instance, the variable $myaddr$ occurring in the
concrete stack is related to $martians$ (invalid IP addresses) as: $myaddr \notin martians$. Note
that the check operation only partially implements the requirements in order to simplify
4.5 Discussion

The split verification methodology enables us to reduce the complexity of proof effort by allowing us to perform interactive theorem proving at the level of primitive operations and then promoting primitive operation-level properties to stack-level primary operations by establishing a set of temporal properties on the operation and its related attributes. Temporal properties may be automatically verified against operations model of the stack by employing symbolic or explicit model checking procedures. Our approach reduces the complexity of theorem proving by introducing the notion of promotion conditions that must be model-checked against the operations model.

Owing to the layered structure of protocol stacks, if the number of compositional layers in a stack is bounded by a constant (this would typically be the case), then the state space and transitions complexity for the stack operations model is polynomially bounded (from Theorem 4.2.1 and Theorem 4.2.2).

One way to control nesting depth in the compositional hierarchy is to view a composite operation as a single operation and perform theorem proving directly on the composite operation. Obviously, this increases the theorem-proving complexity. As a result, we have a trade-off between complexity of model-checking and theorem proving complexity, in presence of nested guarded compositions.

In constructing the stack operations model, we are only considering the compositional structure of the protocol stack and some essential attributes of operations. The model would become significantly more complex if it represents the full implementation details of the stack.

Our approach enables us to subject only a portion of the stack to formal verification and also to perform verification in an incremental fashion. If an operation is modified, we could rerun operation-level verification for only that operation, followed by verification of promotion conditions for that operation. Therefore the cost of proof maintenance would be relatively low for localized changes in the stack.

One limitation that we faced was our inability to translate complex transformations (e.g.,
Figure 4.4: Abstract Z specification of security properties for IP address verification
### Figure 4.5: Concrete IP address verification operation

Checksum, hash, other cryptographic operations) into Z. We overcome this limitation by axiomatically introducing important properties of the transformations into the Z logic system. An example of the axioms that we used for keyed message digest (HMAC) algorithm is given in Figure 4.7. The set of messages are defined using given type `Message`. The `MacFn` is defined as the set of total injective functions that map messages to numerical hash values. The injective definition makes them perfect hash functions. The `KeyedMacFn` maps keys to hash functions.

We assume that protocol stack is constructed in user-space as this simplifies the stack construction and allows us to use standard language constructs without the concerns of
Figure 4.6: The retrieve relation and refinement theorems for the IP address verification

kernel-mode programming issues. In standard kernel-mode processing, the typical overhead of packet processing would involve a minimum of three context switches since processing would be typically done in three contexts, namely, interrupt, kernel thread, and application thread. If we move the protocol stack to user-space this overhead is not increased, we may still retain this threading model and have the packets processed in three contexts, namely, interrupt, basic kernel-level processing, and user-space processing.
[Message]

\[MacFn: \mathcal{P}(\text{Message} \times N \mapsto N)\]

\[\forall fn: MacFn; m1: \text{Message}; m2: \text{Message}; s1: N; s2: N; mac1: N; mac2: N\]

\[. \quad m1 = m2 \land s1 = s2 \iff fn(m1, s1) = fn(m2, s2)\]

\[KeyedMacFn: N \rightarrow MacFn\]

\[\forall k1: N; k2: N \cdot k1 = k2 \iff KeyedMacFn(k1) = KeyedMacFn(k2)\]

Figure 4.7: Axiomatic specification for HMAC algorithm