Chapter 5

Trusted Subject Architecture Implementation

In this chapter we elaborate on implementing the message filter model and the aggressive scheduling scheme under a trusted subject architecture. We begin by discussing a trusted subject architecture for multilevel object-based systems. This is followed by the description of some data structures and various scheduling algorithms. We finally present some theorems and their associated proofs to demonstrate that these scheduling algorithms preserve integrity and confidentiality.

5.1 Architecture

Figure 5.1 illustrates the trusted subject architecture. As mentioned before, the session manager is the trusted subject in this architecture. It is thus a multilevel process and coordinates single-level (untrusted) message manager processes. The session manager is a long-lived process that is created when a session begins and deleted only when the session eventually terminates. A session manager may create several short-lived message-manager processes. Whenever a write-up message is issued, a message manager process is created to service the request, and it implements the message filtering functions.

The interface between a message manager and its local session manager consists of fork, terminate, and start calls. A fork is issued by a message manager to request creation of a new message manager. A terminate call is issued by a message
manager to its session manager and signals termination. A start call is issued by a session manager to a message manager to initiate the execution of the message manager.

What is the motivation, if any, for the trusted subject architecture and implementation? The main advantage is the simplicity with which the scheduling algorithms can be implemented due to the availability of a trusted subject. A session manager always maintains a global snapshot of a session's tree of computations as they progress. With the help of such a global snapshot (view), it is able to coordinate the various concurrent and implement the scheduling algorithms. As we will see in the next chapter, without such a global snapshot, the coordination of concurrent computations has to be achieved in a distributed fashion and this complicates the implementation of the scheduling algorithms.

The above advantage of using a trusted subject for scheduling does come at a price. We now have to provide assurance that such a trusted subject cannot leak information. We later give a noninterference argument to demonstrate that the session manager cannot leak information while coordinating various scheduling strategies and being exempt from mandatory access control rules.

5.2 Scheduling Algorithms

Recall that the session manager in the trusted subject architecture always has a global snapshot of the tree of concurrent computations as they progress to termination. The availability of such a global snapshot significantly reduces the complexity of implementing scheduling algorithms. In fact, we observe that the implementation of the aggressive scheduling is no more complex than the implementation of the conservative one, and as such the latter provides no significant advantage. Hence, for brevity we discuss here only the aggressive scheme for the trusted subject architecture.
Figure 5.1: A trusted subject architecture with TCB

Before discussing the algorithms in detail, we describe the data structures used by the session manager. Recall from chapter 4 that our approach to synchronizing concurrent computations was based on a multiversioning. Every version of an object when created, is assigned a unique timestamp. The session manager maintains the following data structure to keep track of initial versions.

- **Init-stamp**: This is a global table of timestamps with one entry per level. It identifies the initial version of objects at every level that exists before a session starts. An individual message manager can see that portion of the table which is for levels dominated by that message manager.

The session manager also maintains a tree structure that reflects the progress of the
concurrent message managers forked in a session. Every forked message manager is represented by a node in the tree that contains the following information attributes:

status: this can be one of the following: active, terminated, queued;
level: the level of the message manager;
local-stamp: a local table of timestamp entries with an entry at each level dominated by the message manager and identifying the versions at the level that will be used to process read-down requests;
forkstamp: forkstamp issued by the parent message manager;
parent: pointer to parent message manager;
wstamp: This is timestamp entry indicating the next version that will be written by the message manager;
object: receiving object;
message: message;
p: message parameters;

A message manager's local-stamp vector is initialized in two phases, with the first one undertaken when a message manager is forked and the second one deferred until the message manager actually starts. For a message manager just forked, the first phase entries identify the versions to be read at the levels of ancestors, on the path from the root to itself (i.e., the path in a computation tree for a session). These first phase entries are actually obtained by a message manager from another vector that is passed along by its parent. Such a vector can be seen as one that is incrementally constructed along a path in the computation tree. To do this, every message manager is required to save the timestamps in the vector (astamps) obtained from its parent and on issuing a fork, to reconstruct a new vector to give to its child (see figure 3.5). This newly constructed vector will contain the timestamps from the old vector appended with the write stamp wstamp at the level of the issuing message manager. Finally, in the second phase we obtain local-stamp entries for the levels that did not participate in phase one (this is done in the start-trusted-agg procedure of figure 5.3).
Procedure fork-trusted-agg(level-parent, level-create, forkstamp, astamps)
{
Let parent be the node issuing the fork;
Let child be a new message manager node;

Make child the rightmost child of parent;
child.level ← lub[parent.level, L(O2)];

child.forkstamp ← forkstamp;

%Begin phase 1 of acquiring local-stamp entries
For (every level $l \leq$ level-parent)
do
  initialize child.local-stamp table entries from astamps;
End-For

If in a depth-first traversal of the tree starting at the leftmost path and
until child is traversed, there exists a non-ancestor node, say $n$, with
{$n\.level \leq child\.level$ and $n\.status = \text{active or queued}$}
then child.status ← queued;
else
  start-trusted-agg(child);
end-if
}
end procedure fork-trusted-agg;

Figure 5.2: Session manager algorithm for FORK
Procedure start-trusted-agg(nn)
{
    %Let node nn represent the message manager to be started

    % Complete phase 2 of acquiring local-stamp entries
    % Update timestamps from terminated message managers to the left
    Initiate a depth-first search of the tree until node nn is traversed such that:

    If the level l of a node n traversed is not a level of any of the ancestors of
    nn
    and l < nn.level
    then
        nn.local-stamp[l] ← n.wstamp;
    end-if

    % Update remaining local timestamp entries from the Init-stamp table
    If there exists a level l lower than the level of nn and which is neither
    the level of a node traversed in the tree nor of an ancestor of nn
    then
        nn.local-stamp[l] ← Init-stamp[l];
    end-if

    execute(nn);
}
end procedure start-trusted-agg;

Figure 5.3: Session manager algorithm for START
Procedure terminate-trusted-agg(lmsgmgr, wstamp, forkstamp)
{
    Let term be the node that terminated at level lmsgmgr;
    % Mark this node as terminated
    term.status <- terminated;

    % See if any queued nodes can be started
    Initiate a depth-first traversal of the session tree such that:

    If for every leaf node say leaf, that is traversed to the right
    of term such that leaf.level ≥ lmsgmgr, there exists
    no previously traversed non-ancestor node p with {p.level ≤ leaf.level and
    p.status = active or queued}
    then
        start-trusted-agg(leaf);
    end-if
}
end procedure terminate-trusted-agg;

Figure 5.4: Session manager algorithm for TERMINATE

A high-level pseudocode specification of the session manager algorithms to
implement the aggressive scheduling scheme is shown in figures 5.2, 5.3, and 5.4. The
algorithms make extensive use of the tree structure representing the various message
managers. Let us discuss these algorithms in more detail. They are basically designed
to ensure that the invariant inv-aggressive presented in chapter 4, is never violated.
For easy reference, we give the invariant below:

Inv-aggressive: A computation is executing at a level l only if all non-ancestor
computations, in the corresponding computation tree, with smaller fork stamps at
levels l or lower, have terminated.

Whenever a fork request is received (see the procedure in figure 5.2), the
session manager updates its tree structure by creating a node for the forked message
manager and making it the right most child of the parent node issuing the fork. The
procedure then records the forkstamp for the newly forked message manager that
has been passed on by the parent (i.e., the message manager that generated the fork request). This is followed by the first phase of the initialization of the local-stamp entries. The session manager then checks to see if the forked node can be started immediately. To do so, a depth first traversal of the tree is made starting at the leftmost path until the newly inserted leaf node is reached. If during this traversal we find another node, active or queued, at the same or a lower level, the newly inserted node is queued and thus forced to wait.

The processing of a terminate request begins by updating the status of the node to terminated (as shown in figure 5.4). We then check to see if this termination can release other nodes queued up. In determining this, our invariant leads to the property that any nodes started as a result of a termination have to be to the right of the terminated node and at a equal or higher level (and of course, these nodes have to be leaves in the tree). Thus a depth first traversal of the tree is once again initiated. Now as in the fork case, a leaf node is allowed to execute if and only if required by the invariant. It is important to note that a termination may result in more than one node being started. For example in figure 4.5(c) the termination of message manager node 2 (secret) results in nodes 3 (top secret) and 6 (secret) being started.

Both the Fork and Terminate algorithms utilize a common Start procedure (shown in figure 5.3) by which message managers are started. This procedure is primarily concerned with the completing the update of the local-stamp table entries of the node to be started. Recall from our previous discussion that the first phase of updating the local-stamp entries is achieved at fork time. The second phase is now accomplished from the following sources.

1. Terminated left nodes: For levels dominated by a node's level, and for which timestamps were not obtained from the ancestors, the start algorithm looks to
the subtree of computations to the left of the node to be started. The timestamp of the last written versions at such levels is obtained from the last forked message manager (or rightmost node to the left of the node to be started) which wrote at these levels.

2. **Init-stamp table**: If there are levels for which timestamps could not be obtained from phase 1 or from terminated left nodes, the algorithm then retrieves the timestamps from the global Init-stamp table maintained by the session manager. This is because objects at these levels have not been updated so far in the session. Thus the initial versions of objects that existed before the session started at these levels should be used by the starting message manager. The timestamps in the Init-stamp table identify such versions.

Once all the local-stamp entries have been collected, the message manager is started (executed). Thus once a message manager starts, its node in the tree will have all the timestamps necessary to process read down requests for objects classified below its level. These timestamps are never modified in the local-stamp table after start up. However, the timestamp entry stored in the variable \texttt{wstamp} is dealt with differently. On start, the timestamp is incremented unconditionally before the first write (update) operation and subsequently incremented after every fork request issued to the session manager. Thus the timestamp passed on to the forked children by a message manager will vary. Each value identifies the state of the objects at the level of the message manager as of the time the fork was issued.

**Proof of correctness.**

We now state and prove that the aggressive scheduling algorithms under the trusted subject architecture preserve the invariant \texttt{inv-aggressive}. 
Theorem 5.1 The aggressive scheduling algorithms for the trusted subject architecture maintain the invariant inv-aggressive.

Proof:
Message managers get started only in the body of the fork-trusted-agg and terminate-trusted-agg procedures, with a call to the procedure start-trusted-agg. In the procedure fork-trusted-agg in figure 5.2, the precondition to executing the statement start-trusted-agg when the node child is forked is that there exist no non-terminated node to the left of child, at levels dominated by child. This condition continues to hold after child is started. Whenever this condition holds, the invariant is maintained. A similar precondition holds before after the execution of the statement start-trusted-agg in procedure terminate-trusted-agg of figure 5.4 for every leaf node traversed to the right of the just terminated node in the tree. Thus the invariant inv-aggressive is maintained. □

We now state and show how the invariant inv-aggressive maintains serial correctness under our implementation. We state this as a corollary.

Corollary 5.1 The aggressive scheduling implementation maintains serial correctness.

Proof:
We basically have to show how the correctness constraints 1, 2, and 3 are maintained. For a computation to be dequeued and successfully started, invariant inv-aggressive requires all earlier forked computations at level l or lower, to have terminated. But this is what is precisely required to maintain correctness constraints 1 and 2. The argument for the maintenance correctness constraint 3 is independent of the scheduling algorithm used. The local-stamp table entries collected in the first phase at fork time by child reflect versions identifying the states of objects written at the level of
these ancestors before each successive child in the ancestral path was forked (see procedure \texttt{fork-trusted-agg} in figure 5.2). The second phase entries on the other hand identify latest versions written at lower levels for which there were no ancestors. In summary, the read down operations when assigned versions identified by the \texttt{local-stamp} entries, will read the same object states as in a sequential execution, thus ensuring that correctness constraint 3 is maintained. Thus all the three constraints are maintained and serial correctness follows. \Box

\textit{Proof of termination}

We now prove that the aggressive scheduling scheme terminates. In order to proceed with a proof of termination, we assume that once a method (computation) is started, it runs uninterrupted to completion. Obviously, such an assumption can be valid only if the body of the method contains no errors such as an infinite loop. We assume that there is some time-out mechanism in place, to handle such situations. We argue termination of individual computations by formally stating and proving the lemma below:

\textbf{Lemma 5.1} \textit{Once a computation is started, it is guaranteed to terminate.}

\textit{Proof:} The proof follows from two observations:

1. Whenever a computation issues a \texttt{send} which results in a \texttt{FORK}, it is not blocked, but rather runs concurrently with the receiver computation. Now if a computation only issued forked new computations, it is guaranteed to run to completion and terminate since only a finite number of \texttt{FORK} requests can be issued.

2. Whenever a method issues a \texttt{send} that does not result in a \texttt{FORK}, it will be blocked and in general this could result in a chain of blocked methods. However,
there will always be a method executing and progressing to termination at the end of such a chain, and if are no cyclical send relationships, such a method will eventually resume its blocked predecessor. It follows that any blocked method will be resumed eventually and allowed to run to completion in finite steps. □

We now state and prove formally as a theorem that a session will terminate.

**Theorem 5.2** The aggressive scheduling algorithms for the trusted subject architecture guarantee termination of user sessions.

*Proof:*

A message manager (computation) m can be denied immediate execution only at the moment it is forked. If this happens there has to be at least one non-terminated computation to the left of m in the tree. Now by lemma 5.1 every running computation will eventually have to terminate. Also, we know that every termination causes at least one other queued computation to be started. Thus in finite steps all computations to the left of m will terminate, causing m to be started. We can apply this argument to every queued computation and it follows that the entire session will terminate. □.

### 5.3 Proof of Confidentiality

We now give a confidentiality proof for the trusted subject architecture and implementation. Recall that in this architecture, the session manager is a trusted multilevel process that accepts inputs from different security levels, processes them, and outputs information at various levels. Now since the session manager is the only trusted process in this architecture, it follows that any confidentiality leaks would have to be introduced and traceable to the session manager.
Figure 5.5: The two-step processing cycle of the session manager

Before developing a formal security proof, we first give an intuitive argument for the confidentiality of the session manager. We view the session manager as a black box accepting inputs and producing outputs. More precisely, the inputs are the fork and terminate requests issued by message managers at various security levels and the outputs are the start requests issued by the session manager requesting the start-up of previously forked (and perhaps queued) message managers. We assume that these inputs are the sole means by which information enters the process and the outputs are the sole means by which information leaves the process.

Consider first just two security levels low and high (low < high). A fork request issued by a message manager (computation) at low will form a low-input (low-fork) to the session manager. Now such an input can generate only a high output (high-start), when the fork request is processed. Now consider three levels low, med, and high. The termination of a computation at level med can result only in the start-up of another computation at level med or high. Thus in either case, an input generated at level l and received by the session manager, can produce an output only at levels l or higher. Now if we examine the processing steps of the session manager, we see that it is a repetitive two-step cycle of accepting inputs at a level l and producing the associated outputs at l or higher (as shown in figure 5.5). Thus in general any information flow through the session manager occurs only in an upwards direction in a lattice.
Having given an intuitive argument for confidentiality by demonstrating that information flow is always upwards in security levels, we now develop a more formal and rigorous proof using the theory of non-interference [GM82, GM84]. We consider the reception of inputs as well as the generation of outputs to be discrete events. As in [McC90] we call the events less than or equal to a level \( l \) as belonging to the view of that level, and all other events as hidden from \( l \). The basic idea of noninterference can be stated as follows: A subject \( s_1 \) is said to be noninterfering with subject \( s_2 \) if no action issued by \( s_1 \) can influence the future output of the system to \( s_2 \). An obvious approach to establishing noninterference is to purge all hidden inputs and demonstrate that the events observed in the view for a lower level subject remains unchanged.

We begin with some definitions and formalisms.

**Definition 5.1** We define an event as a triple \((\text{type}, l, \text{tstamp})\) where \( \text{type} \in \{\text{fork, term, start}\} \), \( l \) is the level of the message manager (computation) from which the input originated or the level of the message manager to which an output is directed, and \( \text{tstamp} \) is a timestamp indicating the elapsed time since the occurrence of the last event at \( l \).

**Definition 5.2** Given any security level, \( l \), we define the events at less than or equal to \( l \) as belonging to a set called the view of \( l \) and all other events as belonging to a set called hidden from \( l \).

**Definition 5.3** Given any security level, \( l \), we define the subset of events in the view of \( l \) that are at levels strictly below \( l \) as belonging to the set lower-view.

**Definition 5.4** Given two event sequences \( \beta_1 \) and \( \beta_2 \), we say that they are \( l \)-equivalent (denoted as \( \beta_1 \approx_1 \beta_2 \)) if they contain the same events, in the same relative order, for
levels $l$ and lower (i.e., they contain the same values for the event triples and the events associated with these triples appear in the same relative order in both sequences).

Definition 5.5 Given a sequence $\beta$, we define a purge function $\text{purge}(\beta, l)$ as one that returns the sequence $\beta$ but with all events of the form $(\text{type}, l_e, \text{tstamp})$ removed (purged) whenever $l \geq l_e$.

Given any input sequence $\alpha_1$, which when processed by the session manager produces an output sequence $\beta_1$ (denoted $\alpha_1 \rightarrow \beta_1$), noninterference requires us to show the following: If for every level $l$, $\text{purge}(\alpha_1, l) \rightarrow \beta_2$, then $\beta_1 \approx_1 \beta_2$.

Before proceeding on a formal proof that the session manager is noninterfering, we list our assumptions:

- **Input-totality.** If we view the session manager as a state machine, this assumption states that the session manager (or state machine) can accept inputs in any state. This ensures that the session manager is not conveying any information by accepting inputs.

- **Input-output atomicity.** This assumption requires the session manager to accept an input and produce the corresponding outputs, if any, atomically. In other words, in the interval between the acceptance of an input and the subsequent processing and generation of the corresponding outputs, the session manager cannot be interrupted, especially by other inputs.

The assumptions of input totality and input-output atomicity may seem at first to be irreconcilable. After all, if the session manager cannot be interrupted in the interval between the acceptance of an input and the production of the corresponding output, how would it be capable of accepting other inputs that come within such an
interval? Assume for a moment that inputs arrive at the session manager boundary synchronized with clock ticks that are a constant interval apart. It is important to note that such an interval can be chosen to deal with worst case arrival rates of inputs. The session manager is required to accept an input at a clock tick, and produce all the corresponding outputs, before the next clock tick. In other words, at every clock tick, the session manager is ready to accept an input, and within clock ticks cannot be interrupted to accept other inputs.

In the above model, given an input, we require that the corresponding outputs be produced within the same interval. In other words, the outputs cannot spill over to time intervals between subsequent clock ticks. However, one needs to approach the implementation of this requirement with caution. In particular, the timing of the outputs within an interval should not be used to build a channel. Hence we require the scheduler to hold off all outputs until the expiration of the interval. Upon expiration, the outputs are delivered as a batch to a lower level subsystem or operating system.

The realization of the input-output atomicity assumption also requires that the tree data structure implementation utilized by the session manager be an "ideal" one. By ideal we mean that the elementary data structure operations such as the insertion and deletion of nodes in the tree are implemented in such a way that their timing cannot be exploited for covert timing channels. In particular, tree operations should be completed within a clock tick. For if this were not the case, a high user's computation can maliciously cause the tree to grow to a considerable size by causing a lot of nodes to be inserted into it. A low-level computation generating fork requests may now experience observable delays due to the increased time taken by the session manager to update and manage the tree.

A possible solution to deal with the above scenario would be for the TCB to do the tree operations at random intervals. An approach that pursues a similar idea
to address hardware timing channels is based on the technique of fuzzy time [Hu91]. Fuzzy time techniques reduce the bandwidth of timing channels by adding noise to all sources of timing information and by ensuring that inputs and outputs are delivered at random intervals. We do not consider such solutions as they would take us beyond the scope of this thesis.

We now formally state and prove that the session manager is noninterfering.

**Theorem 5.3** In scheduling various concurrent computations, the session manager process is noninterfering.

**Proof:**

The proof is by induction on the length of the input strings accepted by the session manager.

**Basis:** Consider the basis with input strings of length 1. It follows that by accepting only one input at a single security level, say $l$, the corresponding outputs will be at $l$ or higher. This does not influence the outputs at the lower-view of $l$ and it follows that the session manager will be noninterfering. Thus the basis holds trivially.

**Inductive Step:** For the induction hypothesis assume that for all input strings of length $n$, the session manager is noninterfering. For the inductive step, consider any given input string, say $\alpha$, of length $n + 1$ which produces interference. Let $\delta$ be the
\((n + 1)^{st}\) input in this string (see figure 5.6). Also, let this interference be observed at level \(\tau\). By this we mean that there is at least one input at \(\tau\), and the purge\((\alpha, \tau)\) will cause the outputs in the lower-view of \(\tau\) to change. Now let the outputs generated by the scheduler after the reception of the input \(\delta\), belong to the set \(\theta\) with the individual outputs in the set denoted as \(\theta_1, \theta_2, \ldots, \theta_k\). From our earlier discussion on the two-step processing cycle of the session manager, it should be clear that the levels of the individual outputs in the set \(\theta\) dominate the level of \(\delta\) (we denote this as \(l(\theta_j) \geq l(\delta), \forall j, j = 1, \ldots k\)).

Now consider all inputs preceding \(\delta\). Note that an input at a level can only interfere with outputs in the lower-view of the level. So let us pick the most recent input, say \(\iota\) at level \(\tau\), that could interfere with outputs in the set \(\theta\) that are in the lower-view of \(\tau\). We must must now look at the interaction of \(\iota\) and \(\delta\). There are two cases of the input \(\delta\) that we must now consider. The first one being the case where \(\delta\) is a fork input event, and the second where \(\delta\) is a term input event.

For the the first case where \(\delta\) is a fork, let us analyze the procedure fork-trusted-agg in figure 5.2. In this procedure, we determine if the forked computation generating \(\delta\) can be immediately started or if it has to be queued for future execution. This is done by initiating a depth-first traversal of the session tree starting at the leftmost path and ending at the node representing the computation. Now in general, a computation \(f\) is denied immediate execution (startup) only if such a traversal encounters at least one nonterminated non-ancestor computation at or below the level of \(f\) before the node for \(f\) is traversed. Also, it follows from the requirements of serial correctness that the computation generating the input \(\delta\) would have to be to the right of the one generating \(\iota\). (If this were not the case, the computation that would generate \(\iota\) would
be suspended, and as such, there would be no input \( t \) before \( \delta \). Hence, during the depth-first traversal, the computation associated with \( t \) would be encountered before the computation generating \( \delta \). Now if we purge the computation that generated the input \( t \), the outcome from the traversal of the tree will be unaffected. In other words, if the computation associated with \( \delta \) was denied execution, or allowed to start, this will continue to be the case after the purge. In other words, the output generated by the scheduler in response to input \( \delta \) would remain the same for output events in \( \theta \) at the lower-view of \( \tau \). Also, the purge will not generate any additional outputs in \( \theta \). We can thus conclude that these sequences do not lead to any interference.

Now consider the second case when \( \delta \) is a term event. Once again, it follows from our invariants and the requirements of serial correctness that the computation generating the term event \( \delta \) would have to be the right of the computation generating the high input event \( t \), in any session tree. If this were not the case (i.e., if the low computation was to the left of the high computation), the high computation would not be executing due to serial correctness restrictions and thus cannot generate any high inputs. Now let us look at the procedure terminate-trusted-agg in figure 5.4 to see how terminate input requests are processed. We observe that when a computation say \( t \), terminates, a depth-first traversal of the session tree is initiated to identify potential leaf computations to the right of \( t \), that could be released for execution. A leaf computation in the tree is started only if there exists no previously traversed active or queued computation at or below the level of the leaf computation. Now in a depth-first search, the computation generating \( t \) will be encountered before the one generating \( \delta \). However, the computation generating \( t \) is at a higher or incomparable level with respect to the computation generating \( \delta \), and thus the purge of the former will not affect the outcome of the traversal. Thus there is no interference as the out-
put events in $\theta$ that are generated in response to $\delta$ would remain the same. To put it more precisely, the output events in the lower-view of $\tau$ that are in $\theta$, remain the same.

Thus far, we have shown that the input $\iota$ does not interfere with outputs in the set $\theta$ and in the lower-view of $\tau$. Let us now proceed by purging the input $\iota$ from the original string of length $n+1$ to get string $n'$. We will now get a new output set $\theta'$. The set $\theta'$ may differ from $\theta$ only in that it doesn't contain the outputs of $\iota$ which are at the level of $\iota$ or higher. Thus, events in the lower-view of $\tau$ would remain unchanged in both sets $\theta$ and $\theta'$. Now since we just demonstrated that the input $\iota$ causes no interference, it follows that if the original string of length $n+1$ is interfering, this interference must be also observable in the outputs of the string $n'$ with $\iota$ purged. In other words, this interference must be observable in $\theta'$ which retains all outputs below $\tau$ in the original output set $\theta$. However, the string $n'$ is of length $n$, and if it is interfering, will contradict the induction hypothesis which assumed the session manager is noninterfering for all strings of length $n$. Hence the proof for the induction step. $\square$