Chapter 2

Motivation and Problem Statement

In this chapter we discuss the main problem addressed in this dissertation; the support of secure and efficient write-up actions in multilevel object-based computing. We begin with two motivating examples to illustrate the usefulness of write-up operations. We then discuss the various confidentiality, integrity, and performance tradeoffs involved in supporting write-up operations.

Recall that multilevel security mandates information flow to be always upwards in the security lattice. However, there is no reason to disallow a low-level subject from writing-up to higher levels, as the information flow is from low to high (upwards in the lattice). Such write-up actions are natural and very useful in modeling many applications (as we will illustrate shortly). Unfortunately, an analysis of multilevel systems, particularly databases, would reveal that support for write-up actions is generally absent, or at best weak and ad-hoc. These systems typically implement a restricted version of the BLP $\pi$-property that allows writes only at the level of the subject requesting the write operation.

We may partly attribute the above reluctance in supporting write-up actions to a fundamental conflict between confidentiality and integrity [MA91]. This is because the requirements to enforce integrity constraints often result in confidentiality being compromised. Conversely, guaranteeing confidentiality may require tolerating lower degrees of integrity. In conventional databases such as relational systems, the effect of
arbitrary blind write-up operations on integrity is unpredictable and uncontrollable. Thus, there always exists the potential for a low-level subject to obliterate higher-level data. Now in the object-based framework, we cannot foresee a similar threat to integrity. Why is this? Because if objects can communicate solely through messages, the properties of encapsulation and information hiding will ensure that an object state is updated only in controllable ways. On receiving a message from a lower level, a high-level object can exercise complete control over how, and if, its state should be updated. It may choose to reject the message request by not invoking the corresponding method. If the message is accepted, the method invoked has precise semantics known a priori.

Thus the objective of supporting integrity preserving write-up actions seems tenable within the object-based framework. Unfortunately, this convenience does come at a price. Ironically, the very feature of objects (the ability to incorporate well-defined semantics with operations) poses confidentiality leaks. In the object framework, operations are no longer primitive read's and write's, but complex and abstract, and taking varying amounts of processing time. As we will elaborate shortly, this can cause signaling channels when write-up actions are issued [STJ92].

In retrospect, traditional models such as Bell-LaPadula (BLP) address security issues primarily within the realm of multilevel operating systems and not databases. Thus BLP does not associate much semantics with operations; rather, it views operations as basically primitive read's and write's on memory segments. Also, the BLP view of write-up actions boils down to blind writes (i.e., modification of existing higher-level data is not permitted while its overwriting is permitted). Thus a request from a low-level subject to debit or credit a high-level bank account, cannot be handled by the BLP framework. These restrictions, or should we say limitations, have enabled BLP and its derivatives to abstract away the problem of signaling channels
Figure 2.1: Objects in a payroll database

during write-up. If write-up actions are primitive, and thus implemented by machine language instructions such as STORE, it may be reasonable to assume that STORE operations will take a fixed amount of time independent of the data and address. In reality this assumption is only an approximation to what happens in modern computers. Paging, caching, bus contention, and CPU load, to name a few, modulate the time taken to complete operations such as STORE. Depending on the implementation, such variations can be exploited for timing channels and this has been recognized by the security community [Cip90].

2.1 Motivating Applications and Examples

We now motivate the usefulness of write-up operations accomplished by sending messages up in the security lattice, with two examples.

2.1.1 Payroll Processing Example
Consider a database for payroll applications, that has three objects: EMPLOYEE (Unclassified), WORK-INFO (Unclassified), and PAY-INFO (Secret), with the attributes shown in figure 2.1. Every object is assigned a single level. Weekly payroll processing is initiated by the lower level EMPLOYEE object with the sending of the (a) PAY message to the higher level PAY-INFO object. As the receiver is at a higher level than the sender, an innocuous NIL reply is returned by the message filter (as mandated by the message filtering algorithm, which will be discussed in the next chapter). On receiving the PAY message, the method in PAY-INFO sends a read-down message (b) GET-HOURS, to the lower level WORK-INFO object in order to retrieve the hours worked. This information is retrieved and returned in the reply message (c) HOURS-WORKED. Finally, the accumulated hours for the week is reset to zero by the message (e) RESET-WEEKLY-HOURS.

Another scenario for write-up arises when the child-benefits an employee is eligible for needs to be updated due to an increase in the number of children. Such an update is most efficiently accomplished by a trigger fired in the lower level EMPLOYEE object when the NO-CHILDREN attribute changes. The trigger would result in the sending of a message with the value of number of children, NO-CHILDREN, as a parameter to the higher level object PAY-INFO. The alternative to such a write-up would be that the PAY-INFO object scan the corresponding EMPLOYEE object for such changes, whenever the payroll is computed. However, this alternative imposes a significant performance cost for slow-changing information such as NO-CHILDREN. Further, incorporating such monitoring capabilities into methods lowers the reuse potential of the corresponding objects and classes. We elaborate on these issues in the next example.
2.1.2 Situation Assessment Example

Our second example is in the domain of situation assessment. Figure 2.2 illustrates a tactical situation with four objects POSITION-UPDATE (Confidential), TARGET-LOCATOR (Secret), TARGET-TO-SHIP-DISTANCE (Secret), and ACTION-UPDATE (Top-secret). The object POSITION-UPDATE receives and records periodic updates of the position of an AWACS aircraft. After recording the position, it is reported through the message REPORT-POSITION to the object TARGET-LOCATOR. The object TARGET-LOCATOR locates any targets near the reported position and further sends two messages CALC-DISTANCE and DETERMINE-ACTION. The first message is received by the object TARGET-TO-SHIP-DISTANCES which calculates the distances between ships in the fleet and the identified targets, and in turn reports these distances with the message REPORT-DISTANCE to the object ACTION-UPDATE. The second message DETERMINE-ACTION sent by TARGET-LOCATOR is also received by the object ACTION-UPDATE. Finally, on receiving the DETERMINE-ACTION and REPORT-DISTANCE messages, the object ACTION-UPDATE selects one or more ships or other attack vehicles that are within striking range of the target, and initiates some action.

In this example, the messages REPORT-POSITION, REPORT-DISTANCE, and DETERMINE-ACTION, lead to write-up actions. One could always argue whether the above application could be implemented with read-down operations. But we observe that application areas such as situation assessment, battle management, network monitoring, and process control, have sparked a great interest in active databases. Why? Because in these applications the processing steps involve the monitoring of conditions, and the invocation of time-constrained actions when certain conditions come true. In our example, when the AWACS aircraft crosses over a new coordinate, an update of its position is triggered. This update in turn triggers other
processing steps. Implementing these steps with read-down operations would require extensive polling of low level object states by higher level objects.

The difficulty with polling is that it is not very feasible to determine an appropriate polling window (interval) especially when the interval between triggered events is unpredictable and not constant. An inaccurate polling window resulting from guesswork, can have drastic consequences. For example, if the higher level object TARGET-LOCATOR polls the lower object POSITION-UPDATE for updates on the aircraft's position too slowly, it might miss some vital positions that were covered by the aircraft. Clearly, this can result in potential targets escaping detection and identification. On the other hand, if the POSITION-UPDATE object is polled too frequently, it may be flooded with repetitious read-down requests that waste resources and affect performance. In fact, the object may be so overwhelmed with these requests that it may not be able to keep up with the useful and timely position updates from the aircraft. This again, can result in many targets being missed.

2.2 Write-Up and Confidentiality, Integrity, and Performance Tradeoffs

Having motivated the need for write-up actions, we now discuss the issues, conflicts, and trade-offs involved in supporting such actions in multilevel object-based computing environments. In particular, we highlight the trade-offs between confidentiality, integrity, and performance.

Going back to the basics, let us see what happens when a message is sent to a higher security level for the purpose of initiating some write-up action. Figure 2.3 depicts a message $g_1$ sent from a sender object $O_1$ to a receiver object $O_2$, with the receiver classified at a higher security level. Now in synchronous communication mode, the sender method $t_1$ in object $O_1$ is effectively suspended once the message
Figure 2.2: Write-up in situation assessment

$g_1$ has been sent. The receipt of $g_1$ by $O_2$ will result in the invocation of a receiver method $t_2$.

In the multilevel context, it is clear that the contents of the actual reply from $t_2$ cannot be returned to the lower level receiver method (or object), for doing so would lead to an illegal information flow (in fact, the security kernel and mandatory security rules would prevent such an attempt). A conceptually simple solution would be to arrange for an innocuous reply such as a NIL to be substituted and returned by the kernel. This does not result in any direct illegal information flow from the higher level object, as no information based on its contents is made known to the lower level sender. However, it turns out that the very timing of such a reply has broad implications on confidentiality, integrity, and performance issues.\footnote{Strictly speaking, we do not require any reply (NIL or other) to be returned to a suspended}
To elaborate on the above, consider the following alternate ways to deal with message replies:

- **Option 1:** Return a NIL reply on completion of the method in the receiver object;

- **Option 2:** Return the reply independent of the termination of the receiver method in one of the following ways:
  
  - **Option 2a:** Return the NIL reply after some constant time interval that represents an upper bound for completion times;
  
  - **Option 2b:** Return the reply after some random delay;
  
  - **Option 2c:** Return the NIL reply instantaneously.

With the first option, we have a sequential synchronous execution of methods governed by remote procedure call semantics. Now the time taken for the receiver method to complete is by no means constant or predictable. This can happen for example if

sender. The real issue is not so much the timing of the reply, but rather when a suspended method should be resumed. This can be done irrespective of whether a reply is received. Thus it is only for uniformity and ease of exposition, that we assume the receipt of the NIL reply as a logical point to resume a suspended sender method.
the receiver method sends more messages and completes other subtasks before terminating. Further, the receiver method in the higher level object has complete control over its termination. Thus by varying the completion times, the receiver method can modulate the timing of the reply, and this opens up the potential for a signaling channel.

The second set of options attempts to eliminate the above signaling channel by making it impossible for the delivery of the NIL reply to be modulated by a higher level method. Option 2a imposes a heavy performance penalty whenever the receiver method has terminated and the sender remains unnecessarily suspended, waiting for the constant time interval to elapse. If we adopt option 2b, by randomizing the delay before returning the reply, we are faced with a tradeoff between performance and integrity. This is because if the reply is returned well after the termination of the receiver method, we are again unnecessarily holding up the sender method. On the other hand, if we return the reply too early, that is, before the receiver method has terminated, we have to deal with the concurrent execution of methods.

Concurrent executions introduce synchronization problems that can affect the integrity of the database. In particular, it is essential that the concurrent executions guarantee equivalence to a sequential execution, as in the first option. In other words the updates and reads issued by concurrent methods should have the same effect as when the methods are executed synchronously. When such equivalence can be guaranteed, we say that the concurrent execution of the methods (computations) preserve serial correctness. In the next section, we give some concrete examples that illustrate how concurrency can affect serial correctness. Note that this requirement of preserving serial correctness is entirely dictated by integrity considerations. From a confidentiality viewpoint, there is no need to synchronize these concurrent executions.

We now illustrate a scenario on how the integrity of the payroll database
(see figure 2.1) can be compromised. In this scenario, the application semantics and requirements called for a synchronous execution of methods but the resulting execution was concurrent (asynchronous). Now a sequential synchronous execution will lead to the message sequence \( a, b, c, d, e, f \); while a concurrent execution may produce the sequence \( a, d, e, f, b, c \). When weekly payroll processing is initiated by the sending of the PAY message from the lower level EMPLOYEE (U) object to the higher level PAY-INFO (S) object, a NIL reply is returned to object EMPLOYEE and the suspended method in EMPLOYEE resumes execution. Now it is possible for the RESET-WEEKLY-HOURS message which resets the hours worked to zero, to be received and processed by object WORK-INFO before the message GET-HOURS. Thus the message GET-HOURS will retrieve the reset hours as opposed to the actual accumulated hours, resulting in an erroneous calculation of the weekly pay.

We will demonstrate later in chapter 4, how the required integrity can be achieved by the use of a multiversioning scheme that synchronizes concurrent actions on objects so as to guarantee serial correctness. To see how the multiversioning scheme applies to above the payroll example, the (e) RESET-WEEKLY message would result in the creation of a new version of object WORK-INFO with the reset hours. However, an earlier version of object WORK-INFO that existed before the method in PAY-INFO was invoked, is used to process the (b) GET-HOURS message. Serial correctness is now ensured as the GET-HOURS message now retrieves the intended weekly accumulated hours as in the sequential synchronous execution.

Finally, option 2c above calls for replies to be returned instantaneously. We thus no longer incur the performance penalty that is possible with options 2a and 2b. However, we still have to address the integrity issue, as concurrent computations are now inevitable.

How do the various architectures impact our choice of one of the above options?
Option 1 is inherently insecure in trusted subject architectures. Option 1 is further not implementable in a kernelized architecture as the *-property prevents information flow from a higher level to a lower one, by disallowing write-downs (recall that only a trusted subject is allowed to indulge in such write-downs). Such write-down operations are required to inform lower sender methods of the termination of higher level receivers. Option 2a and 2b are implementable in a kernelized architecture but at the cost of performance and integrity. Option 2c needs to address the integrity issue just as option 2b, but offers better performance than the latter, although as with option 2b, this comes at the cost of managing concurrency.

In summary, synchronous RPC-based write-up actions are not secure in trusted subject architectures, and not implementable in kernelized architectures. Thus our only viable choice is to implement message passing and write-up actions in multilevel environments in an asynchronous fashion. The challenge, and our focus, then is to provide the desired RPC-based semantics for asynchronous abstract write-up actions. This requires appropriate synchronization mechanisms, which themselves have to be secure and implementable.