Secure Nested Transactions

Dominic Duggan
Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07030.
Email: `dduggan@stevens.edu`

Ye Wu*
Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07030
Email: `ywu1@cs.stevens.edu`

In the realm of multilevel databases, confidentiality was defined in terms of noninterference for transactional execution at least twenty years ago. Consider for example the following program, that contains a high transaction T_1 and a low transaction T_2:

int$^{\mathsf{Low}}$ `X, Y, Z;`
T_1^{High}: `lock(X); while (1) ;`
T_2^{Low}: `(lock(X); Z=0;)` $\|$ `(lock(Y); Z=1)`

Preventing the writing of sensitive information to a "low" database variable is insufficient, since the use of locks to synchronize accesses to the database provide a covert channel. In this example, T_1 signals to T_2 by locking `X` but not `Y`. In multilevel databases, this leak is prevented by allowing the low transaction to implicitly pre-empt the high transaction when the latter holds a resource that former requires [1].

More recently information flow control has been investigated in the realm of language-based security, as an end-to-end security property of software systems that can be to some extent checked by compilers [2]. The key insight in this work is that noninterference can be related to the control flow in a program, so that indirect leaks through the control flow may be prevented via a type-based control flow analysis. For example, in the following program, there is an apparent information leak due to the writing to "low" variable `Y` after reading "high" variable `X`:

int$^{\mathsf{High}}$ `X;` int$^{\mathsf{Low}}$ `Y;`
`if (X==0) Y=0; else Y=1;`

The fact that `0` or `1` is written to the "low" variable `Y` depends on whether the value of the "high" variable `X` is `0` or `1`. The leak is prevented by the type system, which only allows writes to "high" variables in a local context where "high" variables have been tested.

Ensuring noninterference in concurrent and distributed systems is still a challenge. For example, *termination leaks* in multi-threaded systems allow simulation of the leak prevented by the type system in the sequential example above:

int$^{\mathsf{High}}$ `X, Y;` int$^{\mathsf{Low}}$ `Z;`
`(X=0; Y=1;)`
$\|$ `(while (X==0); Z=0;)` $\|$ `(while (Y==0); Z=1;)`

* Current affiliation: Tencent, Kejizhongyi Avenue, Hi-tech Park, Nanshan District, Shenzhen, China.

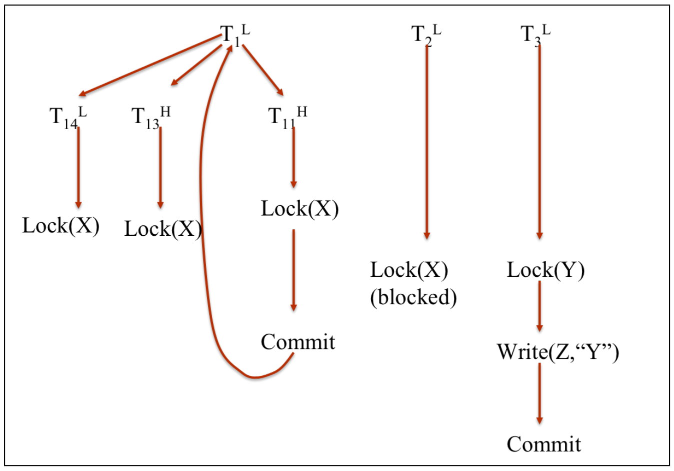

Figure 1. Information Leak with High Child Transaction

Extensions of earlier type systems to prevent such information leaks in multithreaded environments have been proposed, but these extensions miss the point: busy-waiting is one form of synchronization between threads, and it is not clear how to prevent information leaks once synchronization is allowed between threads of different security levels.

Nested transactions were introduced [3] to provide a transactional underpinning to remote procedure call chains. We propose a model for allowing high and low processes to coordinate their activities in multilevel secure systems, without leaking information. This model is based on extending the nested transaction model to support transactional coordination between high and low transactions, while ensuring a noninterference property for their interactions. The semantics of nested transactions is extended with *retroactive abort* in order to support this extension.

Although low transactions must necessarily be able to preemptively abort high transactions, it is equally important that high transactions not be able to abort low transactions. We must assume that the abort of low transactions is observable by low processes. This has implications for the design of the transaction system. Assume that threads and transactions are orthogonal, so that a thread may spawn new threads within the transaction within which it is currently executing. Let the "level" of a thread reflect the level of the variables that it has examined in its context. If a thread has tested the value of a high variable, then its level must be high, preventing it from writing to low-level variables. If we have a transaction that contains both low and high threads, and a high thread acquires a lock, then a low thread (outside that transaction) accquiring that lock can implicitly abort the transaction, and in the process abort any low threads within that transaction. Therefore we cannot have low and high threads within the same transaction. Therefore we refer to transactions as being high or low, reflecting that all threads within a transaction must have the same level. This demonstrates that in the flat transactional model, there can be no intermixing of low and high computations in the single transaction, as in the case of scenarios considered for type systems inspired by Volpano and Smith. This is part of our motivation for considering nested transactions.

In some situations, it may be desirable to have high and low threads to collaborate in a transaction. An example is a low thread that needs to authenticate in order to perform a write, but the credentials are high data. We can allow this cooperation, within some limits, if we adopt the semantics of *nested transactions* [3]. Nested transactions allow trees of transactions, intuitively reflecting call trees in systems of remote procedure calls. The root of this tree corresponds to the first procedure call, and in general a child node in the tree reflects procedure calls that arose from the execution of the parent node procedure call. One of the interesting aspects of this semantics is that, for all transactions but the root transaction in such a tree, commit is a tentative operation. Even if a child transaction commits, its parent may choose to abort, and that in turn requires the child transaction tentative commit to be undone. On the other hand, abort of a child transaction does not require the parent transaction to be aborted. Therefore we can allow high and low threads to co-exist safely in a transaction, provided we isolate them in subtransactions according to their level.

Fig. 1 illustrates the challenges that may arise. In this example, the low transactions T_1^L, T_2^L and T_3^L are cooperating with the high transaction $T_{1.1}^H$ in order to create a covert channel that bypasses the level restrictions on information flow. $T_{1,1}^H$ is a child of T_1^L. The high child transaction $T_{1,1}^H$ acquires the lock for the variable X, in order to establish a covert channel to a low transaction. This high transaction commits, releasing the lock on the variable to its parent (since its commitment must be tentative). The low transactions T_2^L and T_3^L attempt to acquire locks on variables X and Y, respectively. T_3^L acquires the lock on Y, prints a message to this effect, and commits. Since it is outside of any other transaction, its effects are now publicly visible. On the other hand, T_2^L is blocked on attempting to lock X, which was originally locked by $T_{1,1}^H$. Were the latter still active, it would be forced to abort by T_2^L and the lock released. However the lock is now held by the parent of the high transaction, T_1^L, even if this low parent is unaware of the lock it has acquired via the actions of its child. To fix this problem, we require that the high child transaction $T_{1,1}^H$ of T_1^L be *retroactively* aborted. This is possible because the effects of any successful transactions cannot be made visible outside a nested transaction until the root transaction succeeds, and the low parent of a high transaction obviously cannot be aware of whether its high child aborted or committed.

Fig. 1 illustrates two other scenarios related to this. Suppose another high child transaction of $T_{1,3}^H$ has inherited the lock (from its parent T_1^L) that was originally acquired by $T_{1,1}^H$. In this case, $T_{1,3}^H$ will be aborted when the low transaction T_2^L attempts to lock X. Suppose on the other hand that the low child $T_{1,4}^L$ of T_1^L has acquired the lock on X that was originally acquired by $T_{1,1}^H$. If we allow T_2^L to pre-emptively abort $T_{1,4}^L$, then this is caused indirectly by $T_{1,1}^H$, so this is a form of abort dependency from high to low transactions that we are claiming to avoid. In this case, we can say that the low transaction $T_{1,4}^L$ is oblivious of the fact that the lock has been acquired due to inheritance and anti-inheritance from a high sibling, and this amounts to a scenario where a low transaction is blocked due to a resource being held by another low transaction. In this case, there is no information leak and the participation of high transactions in the overall computation is unknown to the low transaction.

We have formalized a model of secure nested transactions with retroactive abort. Since the goal is to show an absence of security leaks due to synchronization between processes, our model is formulated as a process calculus, an extension of the pi-calculus with transactions. In this framework, we are able to use notions of observational equivalence for processes to verify noninterference in the context of distributed processes with synchronization. We are able to formulate and prove a non-interference result: High transactions are unable to leak information to low transactions, since the former are indistinguishable from stopped processes to the latter. Full details are available in the technical report [4].

References

[1] V. Atluri, S. Jajodia, and B. George, *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.

[2] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, 2003.

[3] E. B. Moss, "Nested transactions: An approach to reliable distributed computing," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[4] D. Duggan and Y. Wu, "Security correctness for secure nested transactions," Stevens Institute of Technology, Tech. Rep. 2011-4, May 2011, http://www.jeddak.org/Results/Stevens-CS-TR-2011-4/.