# Transactional Correctness for Secure Nested Transactions

**(Extended Abstract)**

Dominic Duggan Ye Wu
Stevens Institute of Technology
dduggan@stevens.edu

## Abstract

Secure Nested Transactions are an adaptation of traditional nested transactions to support the synergy of language-based security and multi-level database security. They have application in security for enterprise applications, where transactional semantics are a critical feature in middleware systems. This article considers correctness in terms of transactional properties for secure nested transactions. Correctness is expressed in terms of a labeled transition system, the TauZero calculus.

***Categories and Subject Descriptors*** CR-number [*subcategory*]: third-level

***General Terms*** Nested transactions, language-based security.

***Keywords*** Transactions, semantics, serializability.

## 1. Introduction

Security is a hard problem, one that cuts across many disciplines. Providing a yardstick for the correctness of secure systems is equally hard. Information flow control has a long history as a mechanism for stating end-to-end security policies. Noninterference and other properties are stated for the correctness of information flow in secure systems.

Information flow control was investigated for database systems in the context of multilevel database systems. One avenue for investigation has been the interaction of information flow with transaction processing in multilevel databases. The particular issue of interest has been the potential for transactions with "high" security level to implicitly signal to transactions with "low" security level using the synchronization mechanisms provided to ensure proper isolation levels for concurrent transactions. In the example in Fig. 1(a), if transactions $T_1^{\mathsf{High}}$ and $T_2^{\mathsf{Low}}$, which have high and low security levels respectively, use locking to synchronize access to "low" variables $X$ and $Y$, then even though $T_1^{\mathsf{High}}$ may be unable to write to these low variables, it may signal to $T_2^{\mathsf{Low}}$ by locking one variable and not the other. In this example, $\|$ denotes parallel composition, and `X` and `Y` are both low security variables. The high transaction implicitly sets `Z` to 1 by locking `X` and not `Y`. Why might the two transactions need to synchronize on variables at all? The high transaction may be monitoring a database being updated by low transactions, and the isolation property of transactions dictates that it should only be able view consistent states of the database, not an intermediate inconsistent state due to other uncompleted transactions.

This problem has been fairly extensively researched, and for at least one or two decades, it has been understood that in a situation such as this, a low transaction should be able to preemptively abort a high transaction that holds locks that it requires to proceed [1]. This avoids *termination leaks* such as exemplified above.

```
int^Low X, Y, Z;
T1^High: lock(X); while (1) ;
T2^Low: (lock(X); Z=0;) || (lock(Y); Z=1)
```

(a) Transaction Processing

```
int^High X; int^Low Y;
if (X==0) Y=0; else Y=1;
```

(b) Sequential Programs

```
int^High X, Y; int^Low Z;
(X=0; Y=1;)
|| (while (X==0); Z=0;) || (while (Y==0); Z=1;)
```

(c) Concurrent Programs

**Figure 1.** Implicit Information Leaks

More recently information flow control has received attention in programming language security. Here much of the focus has been on reasoning about implicit control flow by relating control flow to data flow. For example, the program in Fig. 1(b) is rejected by type-based security analysis, since it would allow the setting of the "low" security level variable `Y` based on the value of the high security level variable `X`.

*Secure nested transactions* comprise an extension of transactions that synthesizes secure transaction processing, as exemplified by multilevel databases, and programming languages security.

Why should we care about synthesizing these two strands of research? The motivation is the interaction of language-based information flow control and concurrency. If we simply add the ability to fork concurrent threads in a sequential language, we allow leaks that subvert the information flow guarantees provided by type-based analysis. This is demonstrated by the example in Fig. 1(c). It is certainly possible to extend type systems for sequential programs to prevent low security level threads to depend on the termination behavior of high security threads [19, 5, 18]. However this misses the point, as demonstrated by the example in Fig. 1(c), that there is a need for synchronization between threads at different security levels. If mechanisms such as locks are not provided, then applications must implement their own using test-and-set and busy waiting.

The obvious synthesis of these approaches is to have each thread in a concurrent programming language execute as a transaction, and allow "low" threads to pre-empt "high" threads that hold resources that the latter requires. This is an obvious idea, and simple to state. However if we assume that transaction aborts are now visible, either via language primitives for interrogating the state of transactions, or via the scheduler, then each transaction must have just one security level, High or Low. This is analogous to the situation in multilevel databases. However this rules out a class of examples that have

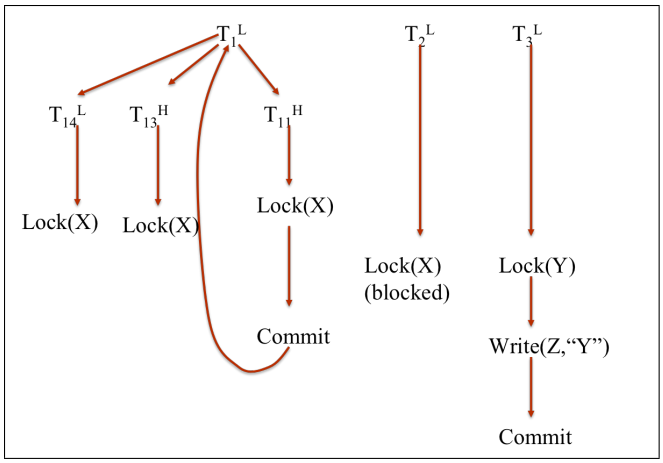



**Figure 2.** The need for retroactive abort

been the raison d'etre for language-based information flow control, *viz.* situations where low-level programs wish to test high-level variables and then perform high-level actions in a temporary high context, before resuming execution as low-level code. In the "flat" transaction model, there is no facility for allowing this. Middleware environments such as Java Enterprise Edition allow transactional RPC calls to start new transactions that are independent of any transaction for the caller bean. However if the caller transaction subsequently aborts, we are left with an "orphan" transaction that by all rights should not exist. The preferred behavior is that the callee should execute as part of the same transaction as the caller. The problem if one is concerned with information flow control is that abort of the callee can force abort of the caller transaction. Therefore low security code cannot call into high security code. Ever.

Nested transactions were proposed by Moss [16] as an extension of the flat transaction model to support transactional remote procedure calls. The nesting of transactions is intended to model a call tree of nested RPCs. The abort of a transaction forces the abort of all transactions descended from that transaction, undoing that RPC and any effects arising from it. On the other hand, abort of a child transaction does not mandate failure of the parent transaction.

*Secure nested transactions* are an adaptation of nested transactions to support synthesize multilevel transaction processing and language-based information flow control. Secure nested transactions provide the same level of synchronization as provided by transactions, while avoiding implicit information leaks such as termination leaks. They also allow the mixing of high and low parts in transactional computations, as demonstrated by the example in Fig. 1(b). Here secure nested transactions leverage the fact that abort of a child transaction does not force abort of the parent. So, as in the sequential case, a "low" computation may test high variables and perform "high" effects in a temporary "high" context.

Fig. 2 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.

Describing the semantics of retroactive abort imposes some challenges. Recent descriptions of transactional semantics for programming languages [4, 11, 15] describe transactional computations where the underlying transactional "machinery" is hidden in the language semantics. We have two reasons for not adopting this approach. First, our intention is to reason about security properties using techniques of observational equivalence from concurrency theory, since we are concerned about information leaks in potentially nonterminating concurrent execution. Second, rather than providing an operational semantics that effectively suggests a particular implementation of retroactive abort, we decouple the details of state management from the operational semantics using an abstraction of *logs*. One may consider logs as a collection of logical statements describing transaction state, and certain operations in the language are predicated on properties being deducible from the logs. For example, once a transaction has aborted, that property enables the restoration of messages that it has consumed. Logs are also not far removed from practical implementations, and protocols such as two-phase commit can be leveraged to check required log properties during the commit of a root transaction.

Our approach is based on a kernel language that has a straightforward implementation. We term our language $\mathbf{Tau_{Zero}}$. Its description is in three parts: a core language derived from the asynchronous pi-calculus, a global collection of logs, and a context of global names (including channel names and transaction identifiers). We use this language to reason about transactional properties of secure nested transactions.

This language is based on the asychronous pi-calculus, a variant of Milner's pi-calculus that accommodates non-blocking message-passing[1]. However our language is *not* a process calculus. Because all channel names are given global scope, there is no way to reason compositionally about the observational equivalence of processes. Neverthess this language is good enough to provide a trace-based semantics that is useful for reasoning about transactional properties such as serializability.

Furthermore, in other work [7] we have developed a processed calculus called $\mathbf{Tau_{One}}$, and used it to verify the security correctness of secure nested transactions. Moreover we are able to relate computations in $\mathbf{Tau_{Zero}}$ and $\mathbf{Tau_{One}}$, using a notion of contextual constraint entailment that is novel. We provide more discussion of $\mathbf{Tau_{One}}$ in Sect. 6.

Several bodies of work demonstrate how higher-level languages may be compiled to various process calculi, and we regard the translation of higher-level languages, extended with retroactive abort, into our calculus as a worthy topic for further research.

We introduce $\mathbf{Tau_{Zero}}$ in Sect. 2. We describe our operational semantics in Sect. 3. We consider transactional correctness

[1] It is worth noting that asynchronous message-passing alone does not support "write-ups" from low to high processes, because of the possibility of traffic analysis of low messages if they can be consumed by high processes. Therefore our semantics includes a special form of message for supporting such communication, that is handled in a linear fashion by the semantics.

| | | | |
|---|---|---|---|
| $C \in$ Channel Type | $::=$ | $(\overrightarrow{C})^{\ell}$ | Message channel |
| | $\mid$ | $\mathsf{Lock}(\overrightarrow{C})^{\ell}$ | Lock |
| | $\mid$ | $\mathsf{Unit}^{\ell}$ | Unit |
| $\ell \in$ Security Level | $::=$ | $\mathsf{High} \mid \mathsf{Low}$ | |
| $T \in$ Type | $::=$ | $C$ | Channel type |
| | $\mid$ | $\mathsf{Trans}(\ell)$ | Transaction type |
| | $\mid$ | $\mathsf{Event}(\overrightarrow{t}, \ell)$ | Event type |
| $w \in$ Name | $::=$ | $a \mid \overrightarrow{t} \mid k$ | |
| $v \in$ Value | $::=$ | $a \mid x \mid ()$ | |
| $A \in$ Agent | $::=$ | $\overrightarrow{t}\,P$ | Agent process |
| | $\mid$ | $A_1 \mid A_2$ | Composition of agents |
| $P \in$ Proc | $::=$ | $\hat{v}\overrightarrow{v}$ | Send message |
| | $\mid$ | $\check{a}\overrightarrow{x}\,P$ | Receive message |
| | $\mid$ | $(v_1{=}v_2) \rightarrow P_1 [] P_2$ | Internal choice |
| | $\mid$ | $P_1 + P_2$ | External choice |
| | $\mid$ | $\overrightarrow{t}[P]$ | Launch transaction |
| | $\mid$ | $P_1 \mid P_2$ | Fork process |
| | $\mid$ | $(\nu a{:}C)P$ | New channel |
| | $\mid$ | $\mathsf{repl}\ P$ | Replicate |
| | $\mid$ | $\mathsf{stop}$ | Stopped |
| | $\mid$ | $\square$ | Commit or abort |
| | $\mid$ | $\mathsf{await}\ t[\square]\ \mathsf{then}\ P$ | Test status |
| $\square \in$ Status | $::=$ | $\boxdot$ | Commit |
| | $\mid$ | $\boxtimes$ | Abort |
| $V \in$ Env | $::=$ | $\varepsilon$ | Empty env |
| | $\mid$ | $(a : C)$ | Channel decl |
| | $\mid$ | $(t : \mathsf{Trans}(\ell)$ | Transaction decl |
| | $\mid$ | $(k : \mathsf{Event}(\overrightarrow{t}, \ell))$ | Event declaration |
| | $\mid$ | $V_1.V_2$ | Append envs |
| $\mathcal{L} \in$ Log | $::=$ | $\mathsf{true}$ | Empty log |
| | $\mid$ | $\mathcal{L}_1 \wedge \mathcal{L}_2$ | Join logs |
| | $\mid$ | $k{::}\overrightarrow{t}\,\hat{a}\overrightarrow{c}$ | Log send |
| | $\mid$ | $k{::}\overrightarrow{t}\,\check{a}\overrightarrow{c}$ | Log receive |
| | $\mid$ | $k_1 \searrow k_2$ | Mesg exchanged |
| | $\mid$ | $k\ \mathsf{undone}$ | Undone receive |
| | $\mid$ | $k_1 \curvearrowright k_2$ | Lock transferred |
| | $\mid$ | $\overrightarrow{t}\boxdot$ | Trans commit |
| | $\mid$ | $\overrightarrow{t}\boxtimes$ | Trans abort |

**Figure 3.** $\mathbf{Tau_{Zero}}$ Syntax

in Sect. 4. We consider related work in Sect. 5, while Sect. 6 provides our conclusions.

## 2. $\mathbf{Tau_{Zero}}$: A Calculus for Secure Nested Transactions

The $\mathbf{Tau_{Zero}}$ language is a two-dimensional calculus of transactions and dependencies, based on Milner's pi-calculus. We assume a core language of asynchronous message-passing, and we extend this familiar idea with a transactional semantics. Our calculus can be viewed as a formal representative for asynchronous messaging systems that are at the core of modern service oriented architectures, providing a transactional semantics for adding messages to, and removing messages from, message queues. Example systems include Java Messaging System (JMS), Microsoft Queuing System (MSQS) and IBM's MQSeries.

The syntax of the language is provided in Fig. 3. We assume the following spaces of variables and names:

$$\begin{gathered} a, b, c, \ldots, \in \text{Channel name} \\ x, y, z, w \in \text{Variable} \\ \overrightarrow{t} \in \text{Transaction id} \\ k \in \text{Event id} \end{gathered}$$

The only values in the language are channel names, represented by constants $a, b, c, \ldots$. Some channels have special significance in their use as locks: they have the property that they are always released by a transaction, whether that transaction commits or aborts. Channels and locks have security levels. These security levels stratify channels into high and low channels, with high channels only usable by high processes and similarly for low channels and low processes. Locks are similarly stratified into high and low, but *low locks may be acquired by high processes.*

We assume the definition of metafunctions $bn(\_)$ and $fn(\_)$ for computing the set of bound and free names, respectively, in a syntactic term. We also assume the definition of the metafunction $fv(\_)$ for computing the set of free variables in a syntactic term.

Each transaction is identified by a sequence of transaction identifiers $\overrightarrow{t} = (t_1, \ldots, t_k)$ for some $k$. Here $t_1$ is intended to be the root transaction, and the complete path identifies a nested transaction and all of its ancestors. We denote the prefix relation between sequences by $\leq$, so we have:

$$\overrightarrow{t_1} \leq \overrightarrow{t_2} \text{ iff } \overrightarrow{t_2} = \overrightarrow{t_1}.\overrightarrow{t_1'}$$

where the period denotes sequence concatenation. Note that $\overrightarrow{t_1} \leq \overrightarrow{t_2}$ means that the former transaction is an ancestor of the latter. This is used extensively in the sequel.

The semantics of the language needs to track dependencies between transactions. In one dimension of this two-dimensional calculus, the dependency is from parent transactions to child transactions. Failure of the parent transaction forces failure of the child, even if the child has tentatively committed. This is to be consistent with the view that no updates propagate from an aborted transaction, including from any of its child transactions. Therefore a transaction type includes the name of its parent transaction, to record this dependency.

There are richer dependencies in our calculus than parent-child. Even when a high transaction commits and its effects are consumed by other transactions, it is possible for the original high transaction to be retroactively aborted and any high successors be subsequently aborted along with it. This requires that $\mathbf{Tau_{Zero}}$ track dependencies between transactions due to exchange of messages.

Fig. 4 explains why this is necessary for secure nested transactions, but not for classical nested transactions. Fig. 4(a) demonstrates an example where $t_0$, $t_1$ and $t_2$ are sibling transactions. If $t_2$ consumes a message output by $t_0$, then the abort of $t_0$ would mandate the abort of $t_2$, indicating a failure dependency. However the only way that $t_2$ can receive a message output by $t_0$ is if the latter commits. If $t_0$ tentatively commits, then it can only be forced to abort if its parent aborts. But if the latter aborts, then $t_2$ must also be forced to abort. Therefore for classical nested transactions, there is no need to track failure dependencies between transactions (beyond parent-child relationships).

Fig. 4(b) demonstrates how secure nested transactions complicate matters. In this example, $t_0$ is a high transaction that has acquired a lock $a$, and subsequently (tentatively) committed.. The sibling $t_2$ has consumed a message $c$ produced by $t_0$. The parent transaction has not yet committed. If a low transaction now preemptively aborts $t_0$ in order to obtain the lock $a$, this will induce an abort of $t_2$. The failure dependency of $t_2$ on $t_0$ is due to the message produced by $t_0$ and consumed by $t_2$. Meanwhile the intermediate low transaction $t_1$ is (necessarily) unaffected. This demonstrates the need to track dependencies due to message exchanges in the semantics. Specifically these dependencies are recorded in the logs.

Therefore we need to track dependencies to propagate cascading aborts of high transactions (although aborts will only cascade within the scope of a parent transaction that contains the high transactions that abort). In order to track dependencies as a result of

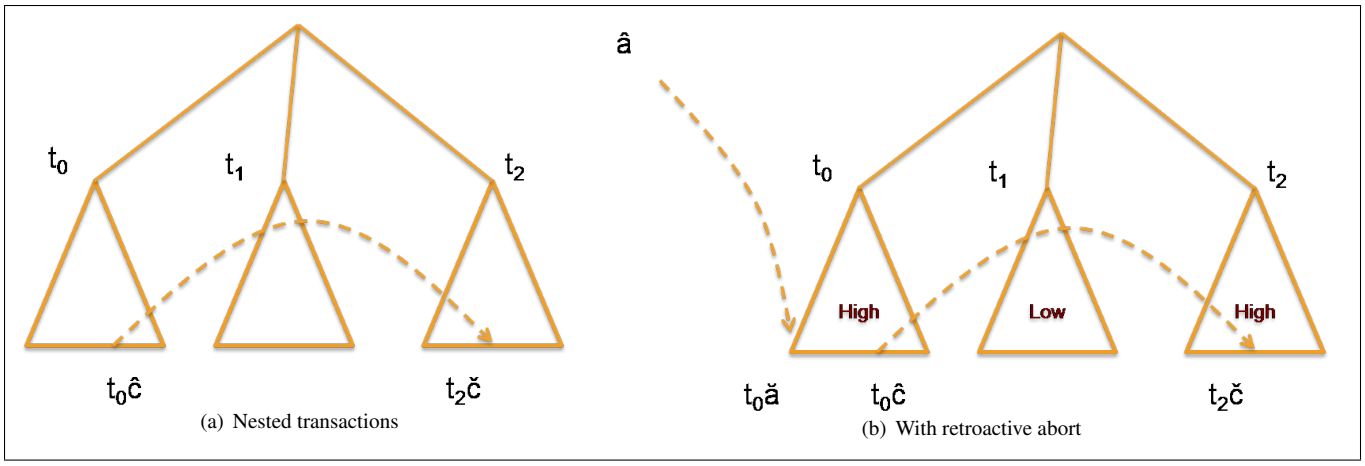



**Figure 4.** Failure dependencies

message-passing and synchronization, we use event identifiers $k$ to uniquely identify significant events in the logs.

A crucial point to note here is that acquisition of a lock by a transaction does *not* induce a failure dependency from transaction owning the lock to the transaction acquiring the lock. In the example in Fig. 4(b), if $\overrightarrow{t_1}$ acquires a lock from $\overrightarrow{t_0}$,then the abort of $\overrightarrow{t_0}$ does not induce the abort of $\overrightarrow{t_1}$. This is due to the restricted access to locks provided to transactions: a transaction cannot duplicate or destroy a lock. The type rules require that locks are generated at the top level, outside any transaction. Once acquired, a log entry records the holding of the lock by the transaction, until abort or commit of that transaction makes it available to other transactions. The release of the lock is guaranteed by the semantics of abort and commit of transactions. In effect we guarantee that locks are handled in a linear fashion: once acquired, a lock is always released. Rather than relying on linear types to statically enforce the linear handling of locks, we rely on the semantics of transactions to enforce this handling.

The syntax of types is provided in Fig. 3. We assume a security type system to prevent information flow leaks, by classifying data as $\mathsf{High}$ or $\mathsf{Low}$. The details of this type system are provided in a technical report [7].

These security levels $\ell$ decorate the types of message channels $(\overrightarrow{C})^{\ell}$ and locks $\mathsf{Lock}(\overrightarrow{C})^{\ell}$, and reflect restrictions on information that can be exchanged as a result of synchronization. Transactions are either "high" or "low," as reflected by their types, and can only have effects (sending and receiving of messages) based on their allowed security level. Whereas in sequential languages, a low thread can raise its security level to high in order to make high side effects, in our language a low transaction must spawn a high child transaction to have high effects. As discussed earlier, if high and low processes occupied the same transaction, a covert channel would be available by having the high process abort the shared transaction.

Although message-passing is asynchronous, we only allow messages to be exchanged between processes of the same security level. Allowing a high security level process to receive a message sent by a low security level process would allow information leaks due to low processes being able to detect contention between high and low processes for such messages. Instead we provide explicit locks to enable synchronization between high and low security level processes. There is a great deal of overlap between the semantics of messages and locks, except in the treatment of lock release for a committed process.

Both messages received and locks acquired by a transaction are recorded in the logs. If the transaction aborts, these message receive and lock acquisition events are undone, releasing the messages and locks back to their original state. As a transaction executes, messages that are intended to be the output of that transaction are "buffered" by limiting their visibility until the transaction commits. Once the transaction commits, those output messages become visible to the parent transaction, and may be received by processes in that parent transaction or descendants of the parent transaction. If a released message is received by a descendant of the parent transaction (or of some ancestor of the sending transaction), then that message receipt is recorded in the log, and a failure dependency established between the sending and receiving transactions.

These failure dependencies take on more significance when we allow transactions of low security level to retroactively abort transactions of high security level. A high security transaction may have acquired the lock, committed, and released a message. This message was received by a (high) sibling, introducing a failure dependency. The committed high transaction is now retroactively aborted, because it still holds the lock required by a low transaction. This abort forces abort of the other transaction that has become failure dependent upon it. Assuming both high transactions have a low parent, the latter is unaware of the changing status of its children.

Locks acquired by transactions must be treated judiciously once those transactions commit. The locks should be released back to the parent transaction. Therefore logs track the acquisition of locks by a transaction, both for undoing their acquisition if the transaction aborts, and for releasing those locks back to the parent if the transaction commits. Unlike messages, locks do not induce failure dependencies. In particular, if a transaction acquires a lock after it was released by a committed transaction, it does not become failure dependent on retroactive abort of the latter transaction.

## 3. Operational Semantics

In this section we consider the operational semantics for $\mathbf{Tau_{Zero}}$. The syntax of the language is provided in Fig. 3. The language includes asynchronous message sending and blocking message receive operations. The message send operation $\hat{a}\,\overrightarrow{v}$ outputs values $\overrightarrow{v}$ on channel $a$, where the latter may be sent as part of another message between processes. The message receive operation $\check{a}\,\overrightarrow{x}\,P$ unpacks a message received on channel $a$ into local variables $\overrightarrow{x}$, and then executes the continuation process $P$. The accent on the channel names is intended to suggest "upload" and "download" re-

spectively. We enrich these basic message-passing operations with both internal and external choice operations ($(v_1{=}v_2) \rightarrow P_1 [] P_2$ and $P_1 + P_2$, respectively), as well as a replication operation $\mathsf{repl}\ P$. The latter is useful for defining recursive processes. We assume that all processes that are ready to input a message have the form

$$\overrightarrow{t} \sum \{\breve{a}\, \overrightarrow{x}\, P\}.$$

In other words, a process may use external choice to select between different input channels. A facility for timeouts could easily be added. A process may launch a new transaction $\overrightarrow{t}\,[P]$ that executes nested within any transaction that encompasses the launching process. The language includes parallel composition and name scoping constructs for each of the forms of constants in the language, as well as a stopped process $\mathsf{stop}$.

The semantics includes a log $\mathcal{L}$. The latter holds information both for reasoning about the status of transactions (e.g. committed or aborted), and also for recovering from the abort of a transactional by undoing any visible effects it has had. The latter take the form of messages consumed or locks acquired during the execution of the transaction. The sending or receiving of a message, and the transfer of a lock, is recorded with a unique event identifier $k$ in the log. The type of this event identifier reflects the transaction in which event occurred. The transaction in turn has an associated security level.

Logs are an important part of controlling the complexity of the transaction calculus. In general configurations in the semantics have components of the form $\overrightarrow{t}\,P$, reflecting that every process $P$ executes with respect to a (nested) transaction $\overrightarrow{t}$. We refer to components of this form as *agents* $A$. The operational semantics are made relatively simple by separating the evolution of process execution from the meta-reasoning about when operational steps are enabled, and what information must be logged to enable the computation.

There are various forms of log rules added during evaluation:

1. A log entry of the form $k{::}\,\overrightarrow{t}\,\hat{a}\,\overrightarrow{c}$ requires the sending of a message or generation of a lock. If the former, the message is sent by a transaction operating within the transaction $\overrightarrow{t}$. If the latter, the type system requires that the lock be generated at top-level, outside the scope of any transaction ($|\ \overrightarrow{t}\ | = 0$).
2. A log entry of the form $k{::}\,\overrightarrow{t}\,\check{a}\,\overrightarrow{c}$ records the receipt of a message or acquisition of a lock by a process executing in the transaction $\overrightarrow{t}$.
3. A log entry of the form $k_1 \searrow k_2$ relates a receive event $k_2$ to the corresponding send event $k_1$. It has several purposes. One is to establish a failure dependency from the sending to the receiving transaction: If the former is aborted, the latter is required in turn to abort. Another purpose is to relate a receive event to the corresponding send event, so that if the latter is undone in the process of aborting a transaction, the corresponding message to be restored is identified.
4. A log entry of the form $k\ \mathsf{undone}$ denotes that the action logged with event identifier $k$ in the logs has been undone. This corresponds to a message receive or lock acquisition event that has been undone because the corresponding transaction has aborted. This type of log entry is used to ensure that message receives are only undone once in the event that a transaction aborts.
5. A log entry of the form $k_2 \curvearrowright k_1$ denotes that the lock acquired in the event labelled with $k_2$ has been released (or "anti-inherited") from a transaction to one of its ancestor transactions, as a result of the former transaction having committed. The lock was then acquired by a descendant of that ancestor transaction, in a lock acquisition event labelled $k_1$. The actual anti-inheritance of locks up the transaction tree is implicit in the committal of ancestor transactions, and fresh log entries for a lock are only added when descendant of one of these ancestors (a "cousin" transaction) acquires the lock. A log entry reflecting the ownership of this lock by the cousin transaction is recorded in the logs with an event identifier $k_1$. It is the counterpart to the $k_2 \searrow k_1$, but where the lock has already been acquired in the transaction tree and now its ownership is being transferred within that tree.

The reduction rules use various judgements to check preconditions by reference to the log:

| | |
|---|---|
| $V, \mathcal{L} \vdash k{::}A$ | Identifiable log entry |
| $V, \mathcal{L} \vdash A$ | Anonymous log entry |
| $V, \mathcal{L} \vdash k_1 \searrow k_2$ | Mesg or lock acquired |
| $V, \mathcal{L} \vdash k_1 \curvearrowright k_2$ | Lock released |
| $V, \mathcal{L} \vdash k\ \mathsf{undone}$ | Action undone |
| | |
| $V, \mathcal{L} \vdash \overrightarrow{t}\ \mathsf{running}$ | Transaction still running |
| $V, \mathcal{L} \vdash \overrightarrow{t}\ \mathsf{aborted}$ | Transaction aborted |
| $V, \mathcal{L} \vdash \overrightarrow{t}\ \mathsf{committed}\ \overrightarrow{t_0}$ | Transaction committed |
| $V, \mathcal{L} \vdash k{::}A\ \mathsf{terminal}$ | Terminal lock ownership |
| $V, \mathcal{L} \vdash k{::}A\ \mathsf{undoable}$ | Undoable receive |
| $V, \mathcal{L} \vdash k{::}A\ \mathsf{transferable}\ \overrightarrow{t}$ | Transferable lock |
| $V, \mathcal{L} \vdash k{::}A\ \mathsf{preemptible}\ \overrightarrow{t_2} \curvearrowright \overrightarrow{t_1}$ | Preemptible trans |
| $V, \mathcal{L} \vdash k_1 \rightsquigarrow k_2$ | Failure dependency |
| $V, \mathcal{L} \vdash k_1 \overset{*}{\curvearrowright} k_2$ | Transfer of ownership |

The first five judgements correspond to simply looking up a log entry, while the remaining judgements are based on inferences drawn from the contents of the log.

The operational semantics of $\mathbf{Tau_{Zero}}$ are specified using reduction rules of the form:

$$(V_1, \mathcal{L}_1, A_1) \Rightarrow (V_2, \mathcal{L}_2, A_2) \quad \text{Internal reduction}$$

and using reaction rules of the form

$$\begin{array}{ll} (V_1, \mathcal{L}_1, A_1) \xrightarrow{\overrightarrow{t}} (V_2, \mathcal{L}_2, A_2) & \text{Visible reaction} \\ (V_1, \mathcal{L}_1, A_1) \longrightarrow (V_2, \mathcal{L}_2, A_2) & \text{Internal reaction} \end{array}$$

The reduction steps correspond essentially to local unfolding in the semantics: unrolling loops where necessary, unfolding conditionals, forking new threads, launching new transactions, etc. The reaction rules correspond to interactions between transactions: message and lock exchange, preemption of all processes in a transaction, as well as committing or aborting a transaction, and testing to see if a child transaction has committed or aborted.

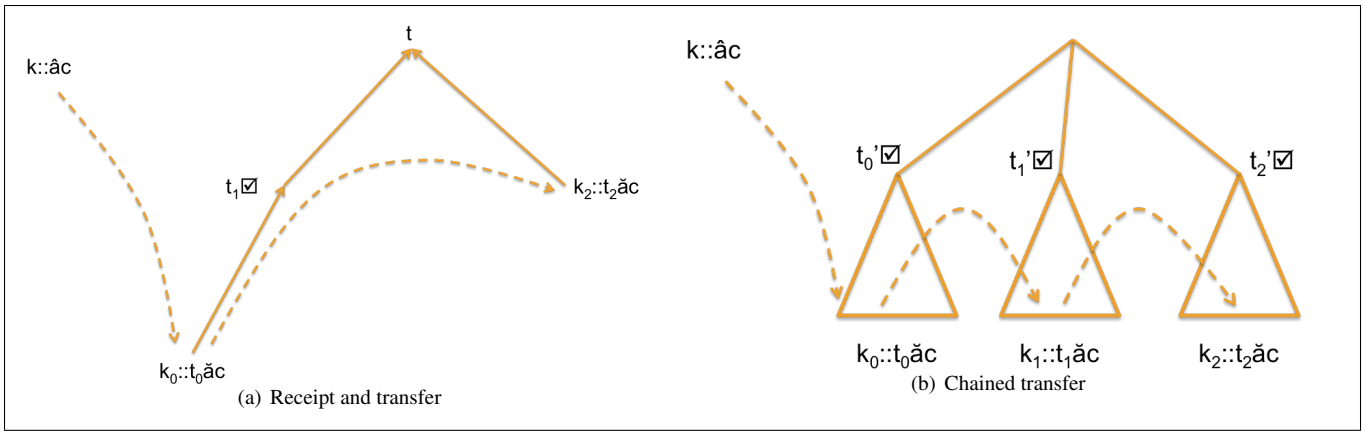
Both reduction and reaction steps potentially modify the context $V$ and the log $\mathcal{L}$. Hence the context and log are outputs from as well as inputs to each reduction step. The visible reaction rules expose the transaction in which the computation step happens. These rules include: a message receive or lock acquisition step (which may consume a message from an ancestor transaction or from the "top level"), a transaction commit step (that releases messages and locks that heretofore had been confined to a child transaction), and a transaction abort step (that releases messages and locks that had been consumed by the transaction before it aborted). We use the notation

$$(V_1, \mathcal{L}_1, A_1) \xrightarrow{[\overrightarrow{t}]} (V_2, \mathcal{L}_2, A_2)$$

to denote a reaction step that may be either visible or internal, with the label $\overrightarrow{t}$ optional.

## 4. Correctness

Define the following notions of repeated reaction steps:



(a) Receipt and transfer (b) Chained transfer

**Figure 5.** Receipt of message/lock and transfer of ownership

$$(V,\mathcal{L},A) \Longrightarrow (V',\mathcal{L}',A')$$
$$\text{iff} \quad \begin{cases} V = V_0, \mathcal{L} = \mathcal{L}_0, A = A_0, \\ V' = V_n, \mathcal{L}' = \mathcal{L}_n, A' = A_n \\ \text{and } (V_j,\mathcal{L}_j,A_j) \longrightarrow (V_{j+1},\mathcal{L}_{j+1},A_{j+1}) \\ \text{for } j = 0,\ldots,n-1 \end{cases}$$

$$(V,\mathcal{L},A) \stackrel{\overrightarrow{t}}{\Longrightarrow} (V',\mathcal{L}',A')$$
$$\text{iff} \quad \begin{cases} (V,\mathcal{L},A) \Longrightarrow (V_0,\mathcal{L}_0,A_0), \\ (V_0,\mathcal{L}_0,A_0) \stackrel{\overrightarrow{t}}{\longrightarrow} (V',\mathcal{L}',A') \end{cases}$$

$$(V,\mathcal{L},A) \stackrel{(\overrightarrow{t_1},\ldots,\overrightarrow{t_n})}{\Longrightarrow} (V',\mathcal{L}',A')$$
$$\text{iff} \quad \begin{cases} V = V_0, \mathcal{L} = \mathcal{L}_0, A = A_0, \\ V' = V_n, \mathcal{L}' = \mathcal{L}_n, A' = A_n \\ \text{and } (V_j,\mathcal{L}_j,A_j) \stackrel{\overrightarrow{t_{j+1}}}{\Longrightarrow} (V_{j+1},\mathcal{L}_{j+1},A_{j+1}) \\ \text{for } j = 0,\ldots,n-1 \end{cases}$$

The first of these corresponds to a sequence of zero more invisible reaction steps (one for which we do not record transaction interactions, such as testing whether a transaction has succeeded or failed). The second of these corresponds to a sequence of zero or more invisible reaction steps, followed by a visible reaction step. The last of these corresponds to a sequence of zero or more visible reaction steps, where there may be invisible reaction steps between each one of these visible steps.

To state a correctness theorem for nested transactions we need some auxiliary definitions. We first define a notion of equivalence between final configurations, based on the output messages that configurations are ready to offer.

**Definition** 4.1. *Define that* $(V,\mathcal{L},A)$ *can output* $\overrightarrow{t}\,\hat{a}\,\overrightarrow{c}$*, written* $(V,\mathcal{L},A) \Uparrow \overrightarrow{t}\,\hat{a}\,\overrightarrow{c}$ *if and only if:*

1. *$a$ is a message channel:* $V(a) = (\overrightarrow{C})^{\ell}$.
2. $(V,\mathcal{L},A) \Longrightarrow (V',\mathcal{L}',\overrightarrow{t_0}\,\hat{a}\,\overrightarrow{c} \mid A')$.
3. $V',\mathcal{L}' \vdash \overrightarrow{t_0}$ committed $\overrightarrow{t}$.
4. $\mathcal{L}' \not\vdash \overrightarrow{t}$ aborted.

Recall that an internal reaction step cannot include a transaction commit or abort operation. We use this to define a notion of final state equivalence between configurations of the operational semantics:

$$(V_1,\mathcal{L}_1,A_1) \approx (V_2,\mathcal{L}_2,A_2)$$

if and only if

1. $V_1(X) = V_2(X)$ for $X \in dom(V_1) \cap dom(V_2)$, and
2. $(V_1,\mathcal{L}_1,A_1) \Uparrow \overrightarrow{t}\,\hat{a}\,\overrightarrow{c}$ if and only if $(V_2,\mathcal{L}_2,A_2) \Uparrow \overrightarrow{t}\,\hat{a}\,\overrightarrow{c}$.

**Definition** 4.2. *Assume* $(V,\mathcal{L},A) \stackrel{\overrightarrow{\overrightarrow{t}}}{\Longrightarrow} (V',\mathcal{L}',A')$*. We say that* $\overrightarrow{\overrightarrow{t}}$ *is a* weakly nested trace *if, for all* $i_1$*,* $j$ *and* $i_2$ *such that* $i_1 < j < i_2$ *and* $\overrightarrow{t_{i_1}} = \overrightarrow{t_{i_2}}$*, we have that* $\overrightarrow{t_{i_1}} \leq \overrightarrow{t_j}$ *or* $\overrightarrow{t_j} \leq \overrightarrow{t_{i_1}}$*. We say that* $\overrightarrow{\overrightarrow{t}}$ *is a* nested trace *if, for all* $i_1$*,* $j$ *and* $i_2$ *such that* $i_1 < j < i_2$ *and* $\overrightarrow{t_{i_1}} = \overrightarrow{t_{i_2}}$*, we have that* $\overrightarrow{t_{i_1}} \leq \overrightarrow{t_j}$*..*

Define a transaction to be an *access* if it contains no subtransactions. Say that a transaction is *proper* if message-passing operations occur only within accesses.

**Theorem** 1 (Serializability). *Suppose* $(V_1,\mathcal{L}_1,A_1) \stackrel{\overrightarrow{\overrightarrow{t}}}{\Longrightarrow} (V_2,\mathcal{L}_2,A_2)$*. Then there is some* $V_3,\mathcal{L}_3,A_3$ *such that:*

1. $(V_1,\mathcal{L}_1,A_1) \stackrel{\overrightarrow{\overrightarrow{t'}}}{\Longrightarrow} (V_3,\mathcal{L}_3,A_3)$.
2. $(V_2,\mathcal{L}_2,A_2) \approx (V_3,\mathcal{L}_3,A_3)$.
3. $\overrightarrow{\overrightarrow{t'}}$ *is a weakly nested trace.* $\overrightarrow{\overrightarrow{t'}}$ *is a nested trace if all transactions in* $A_1$ *are proper transactions.*

As with classical nested transactions, a commit of a non-root transaction is only tentative, since its commit may be aborted by the abort of an ancestor transaction. However we can verify a durability property, that once the root transaction has aborted, then the outputs of a committed transaction are permanent. We verify this by defining an erasure function on processes that removes uncommitted transactions. All operations that affect the logs monotonically add to the logs. The erasure function removes aborted transactions, and removes boundaries for committed transactions, exposes the effects of committed transactions at the "highest" (transaction tree) level where the effects are as yet uncommitted. Once a transaction has been committed all the way to the root, its effects are exposed as top-level messages and locks that cannot be revoked. We omit the details for lack of space.

## 5. Related Work

Birgisson and Erlingsson in [3] present the semantics and implementation for transactional memory introspection (TMI) that supports to enforce security policies under a multiple transaction context. Cohen, Meyden and Zuck [6] develop an access control model

based on Rushby's work [17] to reason about information flow security in their extended system, caused by transaction aborting. All of these works essentially extend work in multilevel databases to software transactional memory.

Transactions have at various points received interest from the programming languages community. The Argus, Camelot and Avalon languages, among several others, incorporated nested transactions to support a transactional semantics for nested RPC [13, 20, 8]. Black et al [4] provide an equational characterization of the ACID properties of transactions. Jagannathan et al [11] provide an operational semantics for transactions in terms of an extension of Featherweight Java [10]. Their calculus supports both nested and multi-threaded transactions, and is agnostic as to whether concurrency control is optimistic or pessimistic. Their main result is the verification of serializability for their semantics. Harris et al [9] provide an operational semantics for software transactional memory abstractions in Haskell, using a monadic semantics to ensure isolation, although without nested transactions. Wojciechowski [21] verifies isolation for a language with nested transactions, extending lock types to ensure that concurrency control is properly attained. Moore and Grossman [15] provide a higher-level operational semantics for nested transactions, one that allows them to investigate several variations on transactional semantics. They use an effect system to ensure isolation of transactions, analogous to the use of monads [9] or extended lock types [21], but abstracting from the operational details of synchronization. This approach is in that respect similar to that of Jaganathan et al. In this framework, they consider various strategies for multi-threaded transactional computation, including scenarios where non-transactional code may access memory concurrently with transactions. Their type and effect system ensures a weak isolation property for such scenarios. A transaction is always isolated from other threads because no thread may access shared memory if another thread is in a transaction.

The concerns for the current article are clearly very different. Our concern is providing an operational semantics for secure nested transactions, in particular with retroactive abort. While the aforesaid approaches are motivated by the need for operational models for software transactional memory, our concern is in reasoning about both transactional and security properties. Thus for example the approach of Moore and Grossman is too high-level for our purposes: They model aborted transactions as transactions that never start in a non-deterministic semantics, which elides many operational details that may be the source of attack vectors. The reliance on type systems in many of the aforesaid approaches to ensure isolation also mitigates their usefulness for our purposes. Indeed, it is already known how to use linear type systems to ensure isolation in concurrent program, without information leaks [12]. The motivation for this work is to consider approaches to controlling information flow that shift the focus from static approaches (beyond a security type system) to the dynamic approach of transactional execution.

Bertino et al [2] consider noninterference for nested transactions. They are principally concerned with the issue of starvation of high transactions in multi-level databases. Rather than preempting a high transaction if it holds a lock that a low transaction requires, they introduce "signal locks" which are essentially callbacks that low transactions use to notify high transactions of updates on shared variables. This allows high transactions to decide what to do when they dynamically discover a race condition. The aforesaid paper extends the approach of signal locks from flat to nested transactions. If a signalled high transaction chooses not to abort, there is no guarantee of serializability between transactions at different levels. If they choose to abort, then there is no longer any guarantee of lack of starvation for high transactions. This work does not consider the issue of locks anti-inherited from high to low transactions, which is the motivation for retroactive abort. In the locking rules in Sect. 4.2 in [2], a transaction is not able to obtain a lock "retained" by another transaction, unless the latter transaction is one of its ancestors. In terms of the example in Fig. 2, transaction $T_2^L$ would be blocked from acquiring the lock implicitly held by transaction $T_1^L$.

## 6. Conclusions

We have described the semantics of secure nested transactions in terms of a language inspired by a process calculus, though it is not in fact a process calculus. In particular, the fact that all channels names are globalized in the semantics makes this inappropriate for reasoning about observational equivalence. In other work [7], we have developed a true process calculus, $\mathbf{Tau_{One}}$, that extends $\mathbf{Tau_{Zero}}$ with local scoping of channel names. The motivation for this work is exactly to reason about security properties, in particular, to draw on results in process equality theory to reason about noninterference independent of transactional properties. This result has profound implications for the system, however. In $\mathbf{Tau_{Zero}}$, all logs are global, and so reaction and reduction rules for computation can interrogate those logs for preconditions that are required for rules to fire. In $\mathbf{Tau_{One}}$, on the other hand, compositionality requires that log entries be distributed amongst processes descriptions. Since reaction and reduction rules no longer have global logs to interrogate, they instead fire in the semantics with logical constraints on the surrounding log context. These constraints propagate upwards through the context of a rule firing, constraining the forms of log entries that can appear in the context.

This "local" semantics is intimately related to the global semantics of $\mathbf{Tau_{Zero}}$. The logical constraints on the firing rules in $\mathbf{Tau_{One}}$ are derived from the preconditions on the firing rules in $\mathbf{Tau_{Zero}}$. Computations in $\mathbf{Tau_{One}}$ are related to those in $\mathbf{Tau_{Zero}}$ using a notion of "constrained contextual entailment." The same coarse trace-based equivalence found in $\mathbf{Tau_{Zero}}$ may be imposed on $\mathbf{Tau_{One}}$. However the point of $\mathbf{Tau_{One}}$ is to use stronger notions of process equivalence, in particular observational equivalence, to reason about security properties independent of transactional properties such as serializabiity.

Our motivation for $\mathbf{Tau_{Zero}}$ is the relative simplicity of its semantics. Specifying reaction and reduction rules in terms of global preconditions (specified on the global logs) is relatively simple to explain. Localizing this with the constrained semantics of $\mathbf{Tau_{One}}$ brings an extra level of complexity, that we prefer to relegate to reasoning about security properties such as noninterference. We claim that $\mathbf{Tau_{Zero}}$ is particularly appropriate for reasoning about transactional correctness because serializability is fundamentally a state-based property. As formulated for databases, the equivalence of schedules, upon which seriailizabilty is based, is derived from the state of the database at the conclusion of different schedules. The coarse-grained equivalence in $\mathbf{Tau_{Zero}}$ only records the interleaving of different transactions. It is in this sense reminiscent of the technique of input-output automata, used to verify many properties of traditional nested transactions [14].

We do not envision any issues with extending this semantics to other synchronization strategies, such as timestamp ordering and multi-version concurrency control. Here we can leverage the fact that logs decouple the form of synchronization from the details of the underlying language, a completely conventional message-passing language. Optimistic concurrency control strategies can be incorporated by redefining the rules for predicates that are used to interrogate the logs, e.g., redefining the rule that currently only makes a message visible outside a transaction when that transaction commits.

## References


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

[2] Elisa Bertino, Barbara Catania, and Elena Ferrari. A nested transaction model for multilevel secure database management systems. *ACM Trans. Inf. Syst. Secur.*, 4:321–370, November 2001. ISSN 1094-9224.

[3] Arnar Birgisson and Úlfar Erlingsson. An implementation and semantics for transactional memory introspection in haskell. In *PLAS*, pages 87–99, 2009.

[4] A. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, lncs, Mumbai, India, 2003. sv.

[5] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002. ISSN 0304-3975.

[6] Ariel Cohen, Ron van der Meyden, and Lenore D. Zuck. Access control and information flow in transactional memory. In *Formal Aspects in Security and Trust (FAST)*, 2008.

[7] Dominic Duggan and Ye Wu. Security correctness for secure nested transactions. Technical Report 2011-3, Stevens Institute of Technology, May 2011. `http://www.jeddak.org/Results/Stevens-CS-TR-2011-3.pdf`.

[8] J. Eppinger, L. Mummert, and A. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1993.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *ACM Conference on Principles and Practice of Parallel Programming*, 2005.

[10] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A core calculus for Java and GJ. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, Denver, CO, 1999. ACM Press.

[11] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 2005.

[12] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 2003.

[13] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3), March 1988.

[14] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufman, 1994.

[15] K. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2008.

[16] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[17] J. M. Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.

[18] A. Sabelfeld. *Semantic Models for the Security of Sequential and Concurrent Programs*. PhD thesis, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden, May 2001.

[19] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 19–21, 1998.

[20] A. Spector and K. Swedlow. Guide to the Camelot distributed transaction facility: Release 1. Technical report, Carnegie Mellon University, 1987.

[21] P. Wojciechowski. Isolation-only transactions by typing and versioning. In *ACM Conference on Principles and Practice of Declarative Programming*, 2005.