# Access Control and Information Flow in Transactional Memory*

Ariel Cohen[1], Ron van der Meyden[2], and Lenore D. Zuck[3]

[1] New York University, arielc@cs.nyu.edu
[2] University of New South Wales, meyden@cse.unsw.edu.au
[3] University of Illinois at Chicago, lenore@cs.uic.edu

**Abstract.** The paper considers the addition of access control to a number of transactional memory implementations, and studies its impact on the information flow security of such systems. Even after the imposition of access control, the Unbounded Transactional Memory due to Ananian et al, and most instances of a general scheme for transactional conflict detection and arbitration due to Scott, are shown to be insecure. This result applies even for a very simple policy prohibiting information flow from a high to a low security domain. The source of the insecurity is identified as the ability of agents to cause aborts of other agents' transactions. A generic implementation is defined, parameterized by a "may-abort" relation that defines which agents may cause aborts of other agents' transactions. This implementation is shown to be secure with respect to an intransitive information flow policy consistent with the access control table and "may-abort" relation. Using this result, Transactional Memory Coherence and Consistency, an implementation due to Hammond et al, is shown to be secure with respect to intransitive information flow policies. Moreover, it is shown how to modify Scott's arbitration policies using the may-abort relation, yielding a class of secure implementations closely related to Scott's scheme.

## 1 Introduction

Multicore architectures have become ubiquitous in the design of microprocessor chips, and they require developers to produce concurrent programs in order to gain a full advantage of the multiple number of processors. Parallel programming, however, is very challenging. It requires programmers to carefully coordinate and synchronize objects that access shared data in order to ensure that programs do not produce inconsistent, incorrect or nondeterministic results. Locks, semaphores, mutexes, and similar constructs are difficult to compose, and their incorrect application may introduce undesirable effects such as deadlocks, priority inversion, and convoying.

*Transactional Memory* (TM) avoids these pitfalls, and simplifies parallel programming by transferring the burden of concurrency management from the programmers to

* This paper to appear in Workshop on Formal Aspects of Security and Trust, Malaga, Spain, Oct 9-10, 2008. ©Springer. The research of the first and third co-authors was sponsored in part by ONR grant N00014-99-1-0131 and NSF Award CNS-0420477. The work of the second author was supported by ARC Discovery grant DP0451529.

the system designers, thus enabling programmers to safely compose scalable applications. Consequently, transactional memory is considered to be a promising alternative method for coordinating objects and numerous new implementations have been proposed recently (see [12] for an excellent survey).

A *transaction* is a sequence of operations that are executed *atomically* – either all complete successfully (and the transaction commits), or none completes (and the transaction aborts). Moreover, committed transactions should be *serializable* – there should be a permutation of the operations of the committed transactions where the operations of each transaction are consecutive and in their original order. Transactional memory allows transactions to run concurrently as long as atomicity and serializability are preserved.

Shared memory systems are often decomposed into security domains, with access control mechanisms used to restrict actions, such as reading and writing memory locations not associated to these domains. This can be for reasons of structural decomposition as well as to enforce an information flow security policy. The latter has been a particular concern in military security applications. An *intransitive noninterference policy* can be viewed as a specification of the permitted causal influences in such architectures.

A great deal of research has focused on construction of multi-level secure systems, but, in practice, such systems continue to be plagued with known insecurities. An alternative that has been advocated is to build a multilevel secure system as a distributed system comprised of single-level systems [14, 15]. Multicore processors offer the potential for such architectures for secure systems to be realised on a single chip. However, appropriate controls on features such as transactional memory will be required to realise this possibility. To our knowledge, the literature on transactional memory has not yet turned to consideration of how access control should be managed in such systems.

In this paper we develop a model of access control in transactional memories, in which transactional memory systems can be seen as extensions of Rushby's [16] access control model that add operations for opening, closing and aborting transactions. We then study the extent to which the theory of information flow in access control systems carries over to the extension. A standard memory system with an access control table can be associated with a "minimal" information flow policy (which is, in general, intransitive). It can be established that a system satisfies the associated minimal information flow policy. We study whether this is also the case for transactional memories, where we focus on different approaches to transactional memory implementation. Specifically, we consider Transactional Memory Coherence and Consistency (TCC) [9], Unbounded Transactional Memory (UTM) [2], and a general scheme for conflict detection and arbitration [17]. Some of these implementations turn out to be secure with respect to the policy associated to arbitrary access control tables, others turn out to be insecure even for the very simple information flow policy involving two agents **H** and **L** with information flow from **H** to **L** prohibited.

Finally, we identify the source of the insecurity to be the ability of one agent's activity to cause another agent's transaction to abort and propose a fix to the classical access control policy that avoids this type of insecurity. We define a generic transactional implementation that incorporates this idea. This result is then used to prove the security of

TCC as well as a modification of Scott's arbitration rules so as to obtain secure variants for all instances of Scott's scheme.

*A Simple Example* To demonstrate the ideas of this paper, consider *Unbounded Transactional Memory* (UTM) that eagerly updates the main memory with new values while maintaining copies of old values in a *transaction log*. A conflict between pending transactions occurs when one tries to read a block that was written by another, or write a block that was read or written by another. The arbitration policy is to abort the younger transaction, i.e., the one that started while the other was already pending.

Suppose a single memory block, $x$, and two security domains, $\mathbf{H}$ (high) and $\mathbf{L}$ (low), where $\mathbf{H}$ can only read $x$ and $\mathbf{L}$ can both read and write $x$. As commonly assumed, the permitted information streams are from each domain into itself and from $\mathbf{L}$ to $\mathbf{H}$. Let $\alpha$ be a trace of the system where $\mathbf{H}$ opens a transaction, then $\mathbf{L}$ opens a transaction, and then $\mathbf{H}$ reads $x$. Now, assume $\mathbf{L}$ attempts to write $x$. According to UTM, since this implies a conflict and $\mathbf{L}$'s pending transaction is younger than $\mathbf{H}$'s, $\mathbf{L}$'s transaction would be aborted. From this, $\mathbf{L}$ would be able to infer that $\mathbf{H}$ has an older pending transaction that read $x$. Consequently, in this case, UTM allows information flow from $\mathbf{H}$ into $\mathbf{L}$, contrary to policy.

The violation of the security policy occurred because of $\mathbf{L}$'s ability to infer information about $\mathbf{H}$ from its *failure* to perform an action successfully, and not, as is usual the case in memory systems, from its ability to read a value written by $\mathbf{H}$. This leads us to the observation that, to avoid such forbidden information flow, one should alter the arbitration policy as to avoid aborting transactions of lower security clients in lieu of actions performed by their higher security peers. As we show in the sequel, in this particular case it suffices to abort the older $\mathbf{H}$ if aborting the younger $\mathbf{L}$ would lead to security violation, and to follow the usual arbitration policy in all other cases.

*Overview* The rest of the paper is organized as follows: Section 2 describes the formal model and recalls the definitions that we use from the theory of information flow security. Section 3 gives a general description of transactional memory system, enhances transactional memory systems with an access control table and proves several implementation of them to be insecure. Section 4 presents a generic secure protocol, and shows security of some transactional memory systems that implement the generic protocol. Section 5 shows how to fix the systems shown insecure in Section 3 so as to be secure. Section 6 reviews the data base literature related to our results. Finally, Section 7 provides some conclusions and discusses future work.

## 2 Model and Access Control

Several different abstract system models have been used in the literature on noninterference. In this paper, we use an *action-observed* model [16], where the observations are outputs received on performing an action. In later sections we refine this model in order to capture specific detail of interest in transactional memory systems.

Let $\mathcal{D}$ be a set of *security domains*, or *agents*, and let $\mathcal{O}$ be a set of *outputs*, or *observations*. An *action-observed* security system (AOSS) is a deterministic machine of the form $\langle S, s_0, A, \mathtt{step}, \mathtt{out}, \mathtt{dom}\rangle$, where

1. $S$ is a set of states (typically, the set of all assignments to some set of variables $V$),
2. $s_0 \in S$ is the *initial state*,
3. $A$ is a set of actions,
4. $\texttt{dom}\colon A \to \mathcal{D}$ associates each action to an element of the set of security domains $\mathcal{D}$,
5. $\texttt{step}\colon S \times A \to S$ is a deterministic transition function, and
6. $\texttt{out}\colon S \times A \to \mathcal{O}$ is a function such that $\texttt{out}(s, a)$ is the output received by domain $\texttt{dom}(a)$ when action $a$ is performed in state $s$.

We write $s \cdot \alpha$ for the state reached by performing the sequence of actions $\alpha \in A^*$ from state $s$, defined inductively by $s \cdot \epsilon = s$ for $\epsilon$ the empty sequence, and $s \cdot \alpha a = \texttt{step}(s \cdot \alpha, a)$ for $\alpha \in A^*$ and $a \in A$.

A *non-interference* policy captures when "actions of agent $p_1$ are permitted to interfere with agent $p_2$," or "information is permitted to flow from domain $p_1$ to domain $p_2$." See Section 6 for a brief survey on the history of non-interference. Formally, a *noninterference policy* is a binary relation $\rightarrowtail$ over $\mathcal{D}$, with $p \rightarrowtail q$ intuitively meaning that "actions of agent $p$ are permitted to interfere with agent $q$." Since a domain should be allowed to interfere with, or have information about, itself, $\rightarrowtail$ is always assumed to be reflexive.

The simplest nontrivial noninterference policy (and the one most studied in the literature) is the one mentioned in Section 1, that comprised of two security domains $\mathbf{L}$ (low security) and $\mathbf{H}$ (high security), with information permitted to flow from $\mathbf{L}$ to $\mathbf{H}$ but not the other way around. Formally, this policy is captured by the (transitive) relation $\rightarrowtail = \{(\mathbf{L}, \mathbf{L}),\ (\mathbf{H}, \mathbf{H}),\ (\mathbf{L}, \mathbf{H})\}$.

As mentioned in Section 1, access control systems can naturally be associated to intransitive noninterference policies (we give the construction at the end of this section). Such policies can be given a number of different semantic interpretations. We use here the notion of *TA-Security* [13] (which avoids some unintuitive information flows allowed in [16]; see [13] for a discussion).

Formally, given sets $L$ and $I$, let $H(L, I)$ be the smallest set $H$ containing $L$ and such that if $x, y \in H$ and $i \in I$ then $(x, y, i) \in H$. Intuitively, the elements of $H(L, I)$ are binary trees with $L$-labeled leaves and $I$-labeled interior nodes. Given a policy $\rightarrowtail$, define, for each agent $p \in D$, the function $\texttt{ta}_p\colon A^* \to H(\{\epsilon\}, A)$ inductively by $\texttt{ta}_p(\epsilon) = \epsilon$, and, for $\alpha \in A^*$ and $a \in A$:

$$\texttt{ta}_p(\alpha a) \;=\; \begin{cases} (\texttt{ta}_p(\alpha), \texttt{ta}_{\texttt{dom}(a)}(\alpha), a) & \texttt{dom}(a) \rightarrowtail p \\ \texttt{ta}_p(\alpha) & \text{otherwise} \end{cases}$$

Informally, the definition builds an operational model of the maximal permitted flow of information, where an action adds to the maximal permitted information of domains with which it is permitted to interfere – the fact that the action occurs, as well as all information available to its domain at the time it occurs.

An AOSS is TA-*secure with respect to* $\rightarrowtail$ if for all $\alpha, \alpha' \in A^*$, and $p \in \mathcal{D}$, if $\texttt{ta}_p(\alpha) = \texttt{ta}_p(\alpha')$ then $\texttt{out}(s_0 \cdot \alpha, a) = \texttt{out}(s_0 \cdot \alpha', a)$ for every $a \in A$ such that $\texttt{dom}(a) = p$. That is, a system is secure if the output of an action returns no more information than the maximal information permitted to be known to its agent.

A simple example of an AOSS is a standard memory equipped with a read/write access control table. Let $\texttt{Loc}$ be a set of memory locations, $Val$ a set of values that these locations may store, and let $\mathcal{R}\colon \mathcal{D} \to \mathcal{P}(\texttt{Loc})$ and $\mathcal{W}\colon \mathcal{D} \to \mathcal{P}(\texttt{Loc})$ represent the locations that each agent is permitted to read and write, respectively. Consider a system in which the set of states is the set of all assignments $s\colon \texttt{Loc} \to Val$, and there are two types of actions: $\texttt{read}_p(x)$ (a read request by agent $p \in \mathcal{D}$ on location $x \in \texttt{Loc}$) $\texttt{write}_p(x, v)$ (a request by agent $p \in \mathcal{D}$ to write value $v$ in location $x \in \texttt{Loc}$.) These actions have the expected semantics: $\texttt{read}_p(x)$ returns the value of $x$ unless $x \notin \mathcal{R}(p)$, in which case it returns $err$. Similarly, $\texttt{write}_p(x, v)$ updates $x$ by $v$ (and returns $ack$) unless $x \notin \mathcal{W}(p)$ (in which case it returns $err$).

We remark that, given the access control structure $\mathcal{T} = (\mathcal{R}, \mathcal{W})$ on such a standard memory, we may define a policy $\rightarrowtail_{\mathcal{T}}$ by $p \rightarrowtail_{\mathcal{T}} q$ iff $p = q$ or $\mathcal{W}(p) \cap \mathcal{R}(q) \neq \emptyset$. (Generally, $\rightarrowtail_{\mathcal{T}}$ is not guaranteed to be transitive.) Intuitively, if $p \rightarrowtail_{\mathcal{T}} q$ then information flow from $p$ to $q$ cannot be prevented, since there is some location that $p$ may write and $q$ may read. Conversely, it can be shown [13] that this relation captures precisely the information flow policies enforced by the access control structure, in the sense that the memory system is TA-secure with respect to a policy $\rightarrowtail$ iff $\rightarrowtail_{\mathcal{T}} \subseteq \rightarrowtail$. In the sequel we examine the extent to which this result generalizes to transactional memories.

## 3 Transactional Memories

Transactional memory systems extend standard memory systems by allowing only for atomic and serializable sequences of operations. Transactional memories vary in their atomicity and serializability policies and in the implementation details by which they guarantee these policies. Consequently, they vary in the data structures they maintain (i.e., set of states) and the algorithms they employ (i.e., "step" function). See [4] for a treatment of issues, which we ignore here.

Assume a set of *clients* that direct transactional requests to a memory system that assigns a value from a set $Val$ to every location $x$ in a set $\texttt{Loc}$ of locations. For every client $p$, let the set of actions $a$ with $\texttt{dom}(a) = p$ (also referred to as *$p$-actions*) be:

- $\blacktriangleleft_p$ – An open transaction request.
- $\texttt{read}_p(x)$ – A request to read from address $x \in \texttt{Loc}$.
- $\texttt{write}_p(x, v)$ – A request to write the value $v \in Val$ to address $x \in \texttt{Loc}$.
- $\blacktriangleright_p$ – A close transaction request.
- $\not\blacktriangleright_p$ – An abort transaction request.

Most current transactional memory implementations assume that each client can read from, and write to, every memory location. Here we take the view that clients are restricted in the locations they may access. Hence, we associate each client $p$ with two subsets of $\texttt{Loc}$, $\mathcal{R}(p)$ and $\mathcal{W}(p)$, that indicate which locations $p$ can read from and write to.

The memory provides a response to each action: $ack$ acknowledges that a non-$\texttt{read}$ has been carried out successfully, a value $v \in Val$ is returned in response to a successful $\texttt{read}$, $err$ signals that the action is invalid, and $aborted$ signals that the transaction within which the action occurs must be aborted. An action is invalid when either it is

a local violation of the transactional sequence, (for example, when a client issues a $\not\blacktriangleright$ and its previous action is also an $\not\blacktriangleright$), or when it attempts to access memory locations that are forbidden. Determining when a transaction is to be aborted depends, however, on the history of the transactional accesses and the transactional policies enforced.

A *conflict* occurs when concurrent transactions access the same location and at least one writes to it. When a conflict occurs, at least one of the participating transactions should be aborted. An implementation has an *eager conflict detection* if it detects conflicts as soon as they occur, and a *lazy conflict detection* if it delays the detection until one of the transactions requests to commit. *Arbitration* policies determine which transaction should abort.

Under *eager version management* the memory is updated with every acknowledged `write` action (which implies that aborts may require a roll-backs), and under *lazy version management* memory updates are delayed until the `write`-ing transaction commits (which entails no roll-backs). Note that eager version management may not be combined with lazy conflict detection.

In [17], Scott studied various notions of conflicts. Let $\prec$ denote the precedence relation on events of a given trace, $e \prec e'$ meaning that $e$ occurred before $e'$ (in the trace). Let $T_p$ and $T_q$ be concurrent (interleaved) transactions of agents $p$ and $q$, respectively. The best known of Scott's conflicts are (1) *lazy invalidation* where $T_p$ and $T_q$ conflict if a write of one transaction may invalidate a read of the other, i.e., if for some memory address $x$, we have $\texttt{read}_q(x), \texttt{write}_p(x, \_) \prec \blacktriangleright_p \prec \blacktriangleright_q$ (Here the read and write can occur in either order); (2) *eager W-R* where $T_p$ and $T_q$ conflict if they have a lazy invalidation conflict, or if for some memory address $x$, we have $\texttt{write}_p(x, \_) \prec \texttt{read}_q(x) \prec \blacktriangleright_p$, and (3) *eager invalidation* where $T_p$ and $T_q$ conflict if they have an eager W-R conflict, or if for some memory address $x$, we have $\texttt{read}_q(x) \prec \texttt{write}_p(x, \_) \prec \blacktriangleright_q$. Scott also studies two arbitration policies. An *eagerly aggressive* policy aborts the transaction that opened first, and a *lazily aggressive* commits a transaction if only if it does not conflict with previously committed transactions.

**Example 1** *Unbounded Transactional Memory* (UTM), proposed in [2], is a hardware transactional memory (HTM) that eagerly updates the main memory with new values while maintaining copies of old values in a transaction log. The description of UTM is outlined in Section 1.

To cast a transactional memory as a AOSS, we assume that the set $\mathcal{D}$ of security domains includes the set of clients, the set $\mathcal{O}$ of outputs includes $Val \cup \{err, ack, aborted\}$, and the set $V$ of variables includes the set `Loc` of memory locations.

Given a sequence of actions $\alpha = a_1 \dots a_n$ and a state $s$, we define the *trace* of $\alpha$ from $s$ to be the sequence $trace(\alpha, s) = (a_1, o_1),\ (a_2, o_2), \dots, (a_n, o_n)$, where $o_i = \texttt{out}(s \cdot (a_1 \dots a_{i-1}), a_i)$ for $i = 1 \dots n$. The trace indicates the sequence of outputs that are obtained for the sequence of actions $\alpha$ when initiated at $s$. We call a pair $(a, o)$ an *event*. We say that $p \in \mathcal{D}$ has a *pending transaction* at $\alpha$ if for some $i \in [1..n]$, $(a_i, o_i) = (\blacktriangleleft_p, ack)$, and $(a_j, o_j)$ is neither $(\blacktriangleright_p, ack)$ nor $(\not\blacktriangleright_p, ack)$ for all $j \in [i+1..n]$, i.e., $p$ has a open transaction which has neither aborted nor committed at $\alpha$. Similarly, $p$ has a *pending aborted transaction* at $\alpha$ if it has a pending transaction and for the maximal $\ell \in [1..n]$ such that $a_\ell$ is a $p$-action, $o_\ell = aborted$.

Some properties of the output function $\texttt{out}$ are relevant for our discussion. As described above, $\texttt{out}$ returns $err$ when an action violates a reasonable transaction sequence or attempts to access forbidden memory locations. More formally, for a sequence of actions $\alpha$ and a $p$-action $a$, $\texttt{out}(s_0 \cdot \alpha, a) = err$ iff one of the following holds:

1. $a$ is $\blacktriangleleft_p$ and $p$ has a pending transaction in $\alpha$;
2. $a$ is not $\blacktriangleleft_p$ and $p$ has no pending transaction in $\alpha$;
3. $a$ is $\texttt{read}_p(x)$ and $x \notin \mathcal{R}(p)$;
4. $a$ is $\texttt{write}_p(x)$ and $x \notin \mathcal{W}(p)$;

(Note that an $err$ output depends solely on local history of agents; If one assumes agents attempt only syntactically "legal" actions, $err$ can be removed.)

An $aborted$ output depends on the implementation details. For simplicity's sake, we require that once an action generates an $aborted$ output, all subsequent actions of the same transaction which do not attempt to abort it, also generate an $aborted$ output. That is, to simplify the exposition, it is assumed that if $p$ has an open aborted transaction in $\alpha$ and $a$ is a $p$-action which is not $\not\blacktriangleleft_p$, then $\texttt{out}(s_0 \cdot \alpha, a) = aborted$. The other cases for which $\texttt{out}$ returns $aborted$ are implementation dependent.

When the transactional memory receives an action, it first checks whether it is syntactically valid, returning $err$ if it is not. It then checks whether an $aborted$ output is due. For all other cases, it outputs $ack$, or some value $v \in Val$ if the action is a $\texttt{read}$ action, which, again, depends on the transactional memory implementation. (For example, in UTM, the value is the last value written, while in other implementations it may be the last value written by a committed transaction). We assume that actions that return $err$ because of an access violation do not update states. That is, that if $a = \texttt{read}_p(x)$ and $x \notin \mathcal{R}(p)$, or if $a = \texttt{write}_p(x)$ and $x \notin \mathcal{W}(p)$, then for every state $s$, $s \cdot a = s$.

Consider a transactional memory, and let $\mathcal{T} = (\mathcal{R}, \mathcal{W})$ be its access control table. As we did above, for a standard memory, we may define the policy $\rightarrowtail_{\mathcal{T}}$ on $\mathcal{D}$ to be the minimal policy consistent with $\mathcal{T}$. More precisely, we have $p \rightarrowtail_{\mathcal{T}} q$ iff $p = q$ or $\mathcal{W}(p) \cap \mathcal{R}(q) \neq \emptyset$. In the case of standard memories, this relation captures precisely the possible flows of information in the system. This proves no longer to be the case when we add the transactional memory structure. In fact, UTM from Example 1, as well as five out of the six combinations of [17]'s conflict and arbitration policies lead to insecure transactional memories, even after we impose access control:

**Theorem 1.** *The following transactional memory protocols are not* TA*-secure with respect to* $\rightarrowtail_{\mathcal{T}}$*:*

1. UTM *as defined in Example 1;*
2. *Protocols with eagerly aggressive arbitration and conflict detection which is lazy invalidation, eager W-R, or eager invalidation;*
3. *Protocols with lazily aggressive arbitration and conflict detection that is either eager W-R or eager invalidation.*

Consequently, the only combination of [17]'s conflict and arbitration policies that Theorem 1 does not cover is that of lazy invalidation conflict and a lazily aggressive arbitration. This is the focus of the next section.

## 4 A Secure Protocol

While standard memories equipped with an access control table $\mathcal{T}$ are TA-secure with respect to $\rightarrowtail_{\mathcal{T}}$ (and, consequently with any policy $\rightarrowtail$ that contains $\rightarrowtail_{\mathcal{T}}$), the results of Section 3 show that, once transactional features are added to a memory, this is no longer the case, since aborts provide a covert channel. All security breaches of Section 3 stem from allowing the output of a one client's action to depend on another's past events it *should* have no access to. Here we propose a remedy to this situation, by restricting the *out*put function so it depends only on the parts of the history that can safely impact the issuer of the action. Our key idea is to equip transactional memories not just with an access control table, but also with an additional control mechanism, that provides a way to constrain this covert channel. After describing the restriction, we present a *generic* protocol that uses the restriction, which we show to be secure. We also show that a well known protocol, TCC, is TA-secure by showing it to be an implementation of the generic protocol. Since TCC employs a lazy invalidation conflict detection and a lazily aggressive arbitration, it shows that the only combination of [17]'s conflicts and arbitrations that is not covered by Theorem 1 has a TA-secure implementation.

Let $\rightarrowtail_{ma}$ (*may abort*) be a reflexive binary relation on $\mathcal{D}$. Intuitively, if $p \not\rightarrowtail_{ma} q$, then in the event of a conflict between a transaction of $p$ and a transaction of $q$, it is $p$'s transaction that should be aborted, else we would have $p$ activity causing an abort of an $q$ transaction, which the relation prohibits.

*The Generic Protocol* We introduce a protocol that uses the relation $\rightarrowtail_{ma}$ to impose the desired properties of `out`. The protocol is "full information" in the sense that it stores, for each client, all the information of actions of clients that may cause it to abort, in the order in which the actions occur. Security of this protocol implies that any implementation of it that allows for less information to be stored is also secure. We refer to this protocol as the *generic protocol*. It is general enough to allow for detection and arbitration of Scott-like transactional memory mechanisms. We show that this generic protocol is secure with respect to a minimal information flow policy derived from the access control and abort restrictions.

The generic protocol is presented as an AOSS. We follow the general model of Section 3, and include, for every client $p \in \mathcal{D}$, an event sequence $Cache_p$ consisting of sequences of events of the form $(a, o)$ where $o \neq err$ and $a$ is a $q$-event for some $q$ such that $q \rightarrowtail_{ma} p$. Thus, the set $V$ of the system's variables consists of:

- For each $x \in \texttt{Loc}$ a variable $\mathit{mem}[x]$ of type $\mathit{Val}$, representing the persistent memory. Initially, $\mathit{mem}[x] = v_0$ for all $x \in \texttt{Loc}$, where $v_0 \in \mathit{Val}$ is some default initial value.
- For every $p \in \mathcal{D}$, a sequence $Cache_p$, initially empty. At each point in time, $Cache_p$ consists of actions (and their responses) of $p$ as well as those of clients that may abort it.

To give operational meaning to the may-abort relation, we construct the implementation so that a client's transactions can be aborted based only on information locally available in the client's cache, and restrict the flow of information into the client's cache to comply with the relation $\rightarrowtail_{ma}$.

We furthermore paramaterize the implementation by means of a *cache policy* whereby each client manages its local cache. This policy, $\mathcal{C}$, is represented by a pair of functions $\mathcal{C}_p = (doomed_p, clean_p)$ for each $p \in \mathcal{D}$, where $doomed_p$ is a boolean function that takes $Cache_p$ and an $p$-action $a$ and returns $true$ iff $p$'s pending transaction should be aborted if $a$ is performed. The function $clean$ defines how each client updates its cache when aborting or committing a transaction. It takes $Cache_p$, and returns a subsequence of it that includes no $p$-events.

The function $doomed$ is assumed to be monotonic: if $doomed_p(C, a) = true$, then so is $doomed_p(C; (a, aborted), b)$ for any $p$-action $b$ other than $\not\curvearrowright_p$. That is, appending further events after a transaction becomes doomed cannot change the fact that it is doomed.

Based on an access control table $\mathcal{T} = (\mathcal{R}, \mathcal{W})$ over the set of locations $\mathtt{Loc}$, a may-abort relation $\rightarrowtail_{ma}$ and a cache policy $\mathcal{C}$, we construct a transactional memory system $TM(\mathcal{T}, \rightarrowtail_{ma}, \mathcal{C})$. The states of the system are based on the variables described above. Fig. 1 describes the steps and output of the generic implementation. For readability, we included only the actions whose output is not $err$ (recall that an $err$ output is a result only of actions that the issuing clients can determine as erroneous). The first column is the action, say $a$. Then second column describes conditions under which $a$ is taken. They are to be read as in a case-statement: the line corresponding to the first condition that holds is to be used. Thus, each can be interpreted as a predicate over states. The third column is $\mathtt{out}(s, a)$ – the output returned when action $a$ is taken from state $s$ that satisfies the associated predicate. The fourth column describes the update to the variables between the current state $s$ and its successor $s' = \mathtt{step}(s, a)$. We use the following two abbreviations: For a set of clients $Q \subseteq \mathcal{D}$, let

$$\mathtt{Update}(Q) := \bigwedge_{q \in Q} Cache_q := Cache_q; (a, \mathtt{out}(s, a))$$

Thus, $\mathtt{Update}(Q)$ is the result of appending the action and its output to all clients in $Q$. For all clients $p \in \mathcal{D}$, $\mathtt{Apply}(Cache_p, \mathit{mem})$ is executed by taking, for each $x \in \mathtt{Loc}$, the most recent occurrence of $(\mathtt{write}_p(x, v), ack)$ in $Cache_p$, and executing $\mathit{mem}[x] := v$. If no such event exists, that $\mathit{mem}[x]$ remains intact. We restrict the set of system states to those reachable from the initial state by means of a sequence of these actions.

The following theorem implies that the *only* way that information may flow between two clients in the generic implementation is by direct reading of written variables and by aborts of one of the client's transactions.

**Theorem 2.** *Given an access control table $\mathcal{T}$, a may-abort relation $\rightarrowtail_{ma}$, and a cache policy $\mathcal{C}$, the system $TM(\mathcal{T}, \rightarrowtail_{ma}, \mathcal{C})$ is* TA*-secure with respect to the policy $\rightarrowtail_{\mathcal{T}} \cup \rightarrowtail_{ma}$.*

An immediate corollary of Theorem 2 is that $TM(\mathcal{T}, \rightarrowtail_{ma}, \mathcal{C})$ is TA-secure with respect to any policy $\rightarrowtail$ that contains both $\rightarrowtail_{\mathcal{T}}$ and $\rightarrowtail_{ma}$.

Theorem 2 can be similarly proved for protocols that record less information than the generic protocol above. For example, if $p$ performs read or write actions on locations not observable by $q$, then such operations need not be recorded in $q$'s cache. Other

| action ($a$) | case (use first that applies) | output | updates |
|---|---|---|---|
| $\blacktriangleleft_p$ | | $ack$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ |
| $\texttt{read}_p(x)$ | $doomed_p(Cache_p, \texttt{read}_p(x))$ | $aborted$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ |
| | $Cache_p$ has $\texttt{write}_p(x, v)$ not proceeded by $\texttt{write}_p(x, -)$ | $v$ | $\texttt{Update}_(\{q : p \rightarrowtail_{ma} q\})$ |
| | $Cache_p$ contains $(\texttt{read}_p(x), v)$ | $v$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ |
| | otherwise | $mem[x]$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ |
| $\texttt{write}_p(x, v)$ | $doomed_p(Cache_p, \texttt{write}_p(x, v))$ | $aborted$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ |
| | otherwise | $ack$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ |
| $\not\blacktriangleright_p$ | | $ack$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ ; $Cache_p := clean_p(Cache_p)$ |
| $\blacktriangleright_p$ | $doomed_p(Cache_p, \blacktriangleright_p)$ | $aborted$ | $\texttt{Update}(\{q : p \rightarrowtail_{ma} q\})$ |
| | otherwise | $ack$ | $\texttt{Apply}(Cache_p, mem)$; $Cache_p := clean_p(Cache_p)$ |

**Fig. 1.** Steps of generic implementation

variants (that still maintain the soundness of Theorem 2, with some modifications to its proof and system definitions) are protocols where the $clean$ functions do not necessarily wipe out the most recent transaction of a client.

Consider now the case of Scott's scheme with a lazy invalidation and lazily aggressive arbitration: a conflict occurs when a transaction that writes to some memory location commits while there is another transaction that had read from this memory location, and it is arbitrated by aborting the reading transaction. Note that [17] is implicitly confined to lazy version management, which implies lazy conflict detection. We denote such a transactional memory system by $M_{Lazy}$. That is, given a set of locations $\texttt{Loc}$, $M_{Lazy}$ is the transactional memory system over the locations $\texttt{Loc}$ in which the states are just sequences of actions, the initial state is the empty sequence $\epsilon$, and the $\texttt{step}$ function is defined by concatenation: $\texttt{step}(\alpha, a) = \alpha a$. The observations in this system are uniquely defined once we specify that the system is a transactional memory system with lazy invalidation conflict, lazily aggressive arbitration, and lazy version management: $\texttt{out}(\alpha, a)$ is the unique output value implied by this specification when the sequence $\alpha$ is followed by action $a$. (We assume here that the output of any read in a transaction, but the first one, is handled by the local cache rather than by access to the main memory.)

An example of a $M_{Lazy}$ system is the Transactional Coherence and Consistency (TCC) system of [9]. There, each client executes its transaction speculatively in its cache, and at commit, updates the memory and broadcasts all the write locations of the entire transaction to the other clients, notifying them about those locations that have been updated. When a client receives the broadcast, it aborts its current transaction if the broadcast indicates that some memory location read in the current transaction had been updated by the transaction of the broadcasting client.

**Theorem 3.** *For each access control table $\mathcal{T}$, the systems $M_{Lazy}(\mathcal{T})$ and $\text{TCC}(\mathcal{T})$ are* TA*-secure with respect to the policy $\rightarrowtail_{\mathcal{T}}$.*

Thus, the one case of Scott's schema where we did not show insecurity is in fact secure.

## 5 Securing the Insecure Implementations

The may-abort relation of Section 4 can also be used to enforce security on transactional memory systems that are not inherently secure, for example, the five schemata that are shown insecure in Theorem 1.

Recall the conflicts studied here (see Section 1). According to the definition of the relation $\rightarrowtail_{\mathcal{T}}$ (Section 3), if $p \rightarrowtail_{\mathcal{T}} q$ then there is a potential for a conflict between pending transactions of client $p$ and client $q$. Since in the case of a conflict one of the transactions must be aborted, it is only reasonable to assume that $p \rightarrowtail_{\mathcal{T}} q$ implies that at least one of $p \rightarrowtail_{ma} q$ or $q \rightarrowtail_{ma} p$ holds.

The arbitration policies determine, in a case of a conflict, which of the conflicting transactions should abort. As we saw, however, some such aborts may lead to security violations. We propose to remedy the situation by altering the arbitration rule, taking into account the $\rightarrowtail_{ma}$ relation. The policies are identical to Scott's when the client selected for abort may be aborted by the other according to the $\rightarrowtail_{ma}$ relation; it makes opposite decision in other cases.

Assume that pending transactions $T_p$ and $T_q$ conflict, and $T_p$ attempts to close. The proposed arbitration policy is:

**eagerly aggressive.** Let $r \in \{p, q\}$ be the client whose transaction opened *first*, and let $\bar{r}$ be the other client. If $\bar{r} \rightarrowtail_{ma} r$, then abort $T_r$, and otherwise abort $T_{\bar{r}}$. That is, if the client whose transaction opened later may abort the one whose transaction opened earlier (and is about to close), then abort the latter's transaction. Otherwise, abort the transaction that opened later (which is consistent with traditional arbitration).

**lazily aggressive.** If $p \rightarrowtail_{ma} q$, then abort $T_q$. Otherwise, abort $T_p$.

We now show that, with these revised arbitration rules, all six combinations of conflict and arbitration policy lead to secure implementations. Given an access control table $\mathcal{T}$, a Scott conflict rule CONF, and a modified arbitration rule ARB with respect to the may-abort policy $\rightarrowtail_{ma}$, let $M(\mathcal{T}, \text{CONF}, \text{ARB}, \rightarrowtail_{ma})$ be the transactional memory system that applies the access control policy $\mathcal{T}$ and makes abort decisions by resolving conflicts generated by CONF according to arbitration rule ARB wrt $\rightarrowtail_{ma}$.

**Theorem 4.** *Suppose that $\mathcal{T}$ is an access control table and $\rightarrowtail_{ma}$ is a may-abort relation such that if $p \rightarrowtail_{\mathcal{T}} q$ then $p \rightarrowtail_{ma} q$ or $q \rightarrowtail_{ma} p$.*

*Then the system $M(\mathcal{T}, \text{CONF}, \text{ARB}, \rightarrowtail_{ma})$ is* TA*-secure with respect to the policy $\rightarrowtail_{\mathcal{T}} \cup \rightarrowtail_{ma}$.*

## 6 Related Work

The notion of noninterference was proposed by Goguen and Meseguer [7] in order to provide an abstract characterization of information flow. The original theory was for

*transitive* security policies (where, if information is permitted to flow from A to B and from B to C, it is permitted to flow from A to C). Intransitive noninterference policies, for which the semantics of [7] is insufficient, are gaining renewed significance in the context of the MILS (Multiple Independent Levels of Security and Safety) approach to high-assurance systems design [1, 18]. This approach envisages the utilization of recent advances in, e.g., the efficiency of separation kernels, to increase the degree of componentization of systems, enabling secure systems to be built from a mix of small, trusted and more complex, untrusted components [15], with global security properties assured from the separation property and a verification effort focused on the trusted components. An intransitive noninterference policy can be viewed as a specification of the permitted causal influences in such an architecture. As we have noted, access control structures in shared memory systems are also associated with implicit noninterference policies, that are generally intransitive.

Haigh and Young [8], generalized the work of [7] to *intransitive* policies. Their theory was refined by Rushby [16], who also presented results showing that for a certain class of access control systems, if the read/write constraints in the system are compatible with an information flow policy then the system is in fact information flow secure with respect to that policy (which is, in general, intransitive). The definitions and theory of intransitive noninterference have recently been clarified by van der Meyden [13], whose definitions we follow here.

A significant body of literature exists on multilevel secure databases, in which the issue of transaction processing has been addressed. The area is surveyed in [3]. Covert channels that are similar to those identified in this paper are known for many of the traditional database transaction processing protocols. Closest to the transactional memory protocols we have considered are the multi-version (corresponding to lazy version management) optimistic schedulers (which, like transactional memory, do not delay requests, but execute them speculatively). Keefe et al [11] discuss a multi-version optimistic protocol with the following rule for aborts: "A transaction attempting to commit is aborted if its read set conflicts with the write set of another transaction that committed after it started." They show that this protocol, is secure for "class 2-SS" transactions, which are transactions that may write to variables of a higher security level, but involve only a single subject, i.e. agent. With respect to Scott's scheme, this amounts to lazy invalidation, but the arbitration rule differs from Scott's rules. Note that it causes unnecessary aborts when the reads all occur after the commitment of the closed transaction. Downing et al. [5] discuss another optimistic protocol that seems to be more closely related to $\textsc{tcc}(\mathcal{T})$.

There are some differences to our work, however. The database literature has concentrated on transaction scheduling on uniprocessor systems whereas the motivation for our study of transactional memory is multi-processor systems. The literature on multilevel database transactions assumes a partially ordered set of security levels, which corresponds to a transitive security policy. In this respect our work is more general in that we deal with intransitive policies, and note that these arise naturally from access control tables and the may-abort relation.

On the other hand, the database literature has considered several issues that we have not attempted to address. These include transactions involving multiple agents, and

transactions for a single agent operating at multiple security levels - we have treated just transactions operating with respect to a single security classification. Distributed transaction processing issues such as atomic commitment protocols have also been studied from the perspective of information flow security. We have assumed here that each transaction executes on a single processor. It would also be interesting to consider such questions in the context of transactional memory.

## 7 Conclusion and Future Work

Mechanisms for shared use of resources in concurrent systems, such as locks and caches, are well-known to be potential sources of covert communication channels. Our results show that transactional memory, while it may help to reduce the requirement for mechanisms such as locks, may well open up new covert channels.

We first extended Rushby's access control model to transactional memory by adding operations for opening, closing and aborting transactions, and obtained a model of access control for transactional memories. We then studied the theory of information flow as applied to this model. Several well-known implementations were shown to be insecure. UTM was found to be insecure even for the very simple information flow policy involving two agents **H** and **L** with information flow from **H** to **L** prohibited. We also examined various combinations of conflict and arbitration functions that were introduced by Scott. Similarly to UTM, five out six combinations that were explored were found to be insecure for the very simple information flow policy that involves two agents.

The first straightforward conclusion from these results is that just extending implementations with an access control table is not sufficient for obtaining a secure system, and further methods and restrictions must be applied. It is worth pointing out that through our research we reviewed many implementations that do not even provide basic means for preventing restricting information flow between clients. Some implementations, e.g., DSTM [10], allow clients to directly abort transactions of other clients, or to modify their local data without any additional mechanism that prevents them from abusing it to signal (and thus pass information) the other clients. We therefore suggest that the issue of security should be considered at the early stages of design of transactional memory.

We have proposed the specific mechanism of adding a "may-abort" relation to the implementation. Based on this idea, we defined a generic implementation that is parameterized by an access control table and a may-abort relation as well as a cache management policy. We showed that all instances of this generic implementation are secure with respect to a policy derived from the access control table and may-abort policy. Using this result, we proved the security of the well known implementation TCC, which employs lazy version management and lazy conflict detection and executes transactions speculatively in the clients' caches. We were also able to propose a modification of Scott's arbitration policies that ensures security of all instances of Scott's scheme, by conditioning aborts to comply with the may-abort policy.

The most natural next step is to consider additional implementations from the perspective of our results. We note that UTM, since it employs eager version management, is not an instance of the generic implementation, which is based on lazy-version man-

agement. It would be of interest to also develop a generic secure implementation that covers variants of implementations based on earger version management. Another direction is to explore other sources that may potentially lead to insecurity in transactional memory, e.g., overflow in HTM.

The field of transactional memory is very open and relatively little formal work has been done. We have studied a particular formal model here, but others are conceivable. In particular, one could take a very different view as to what constitutes the interface of a transactional memory that should be studied from the point of view of information flow. For example, whereas we have considered aborts to be observable and left it open that an aborted transaction might be abandoned (rather than retried), one could consider a level of abstraction at which aborted transactions are automatically retried by the transactional memory system and the outcome of transactions is only visible at the interface once the transaction has successfully committed. Given our asynchronous semantics, the information flow violations that we have identified would probably not occur at this level of abstraction, though they would still be reflected as observable latencies on a timed model. Whatever one's opinion of such matters, our work demonstrates that information flow errors are an issue in transactional memories, and gives insight into how they might be resolved.

Finally, we note that the formal notions of security we have applied are based on an asynchronous model of computation, and do not take timing channels into account. With caches being a key implementation detail of transactional memory, temporally sensitive notions of information flow also need to be considered.

## References


1. J. Alves-Foss, W.S. Harrison, P. Oman, and C. Taylor. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3/4):239–47, Feb 2006.
2. C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.
3. V. Atluri, S. Jajodia, and B. George. *Multi-level secure transaction processing*. Kluwer, 2000.
4. A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *Proceedings of FMCAD 2007*, 11 2007.
5. A.R. Downing and T.F. Greenberg, I.B.and Lunt. Issues in distributed database security. In *Proceedings of Fifth Annual Computer Security Applications Conference*, pages 196 – 203, 12 1989.
6. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT-Press, 1995.
7. J.A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Oakland, 1982.
8. J.T. Haigh and W.D. Young. Extending the noninterference version of MLS for SAT. *IEEE Trans. on Software Engineering*, SE-13(2):141–150, Feb 1987.
9. Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun.

Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

10. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.
11. T.K. Keefe, W.T. Tsai, and J. Srivastava. Database concurrency control in multilevel secure database management systems. *IEEE Trans. Knowledge and Data Engineering*, 5(6):1039–1055, 12 1993.
12. James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.
13. R. van der Meyden. What, indeed, is intransitive noninterference? In *European Symposium on Research in Computer Security*, volume 4734 of *LNCS*, pages 235–250. Springer, Sep 2007.
14. Norman E. Proctor and Peter G. Neumann. Architectural implications of covert channels. In *Proc. 15th National Computer Security Conference*, pages 28–43, 1992.
15. J.M. Rushby and R. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, 1983.
16. John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI international, dec 1992.
17. M.L. Scott. Sequential specification of transactional memory semantics. In *Proc. TRANSACT the First ACM SIGPLAN Workshop on Languages, Compiler, and Hardware Suppport for Transactional Computing*, Ottawa, 2006.
18. W.M. Vanfleet, R.W. Beckworth, B. Calloni, J.A. Luke, C. Taylor, and G. Uchenick. MILS:architecture for high assurance embedded computing. *Crosstalk: The Journal of Defence Engineering*, pages 12–16, Aug 2005.