Safe Serializable Secure Scheduling:
Transactions and the Trade-Off Between Security and Consistency

Isaac Sheff   Tom Magrino   Jed Liu   Andrew C. Myers   Robbert van Renesse

Cornell University   Department of Computer Science   Ithaca, New York, USA

{isheff,tmagrino,liujed,andru,rvr}@cs.cornell.edu

ABSTRACT

Modern applications often operate on data in multiple administrative domains. In this federated setting, participants may not fully trust each other. These distributed applications use transactions as a core mechanism for ensuring reliability and consistency with persistent data. However, the coordination mechanisms needed for transactions can both leak confidential information and allow unauthorized influence.

By implementing a simple attack, we show these side channels can be exploited. However, our focus is on preventing such attacks. We explore secure scheduling of atomic, serializable transactions in a federated setting. While we prove that no protocol can guarantee security and liveness in all settings, we establish conditions for sets of transactions that can safely complete under secure scheduling. Based on these conditions, we introduce *staged commit*, a secure scheduling protocol for federated transactions. This protocol avoids insecure information channels by dividing transactions into distinct stages. We implement a compiler that statically checks code to ensure it meets our conditions, and a system that schedules these transactions using the staged commit protocol. Experiments on this implementation demonstrate that realistic federated transactions can be scheduled securely, atomically, and efficiently.

1. INTRODUCTION

Many modern applications are distributed, operating over data from multiple domains. Distributed protocols are used by applications to coordinate across physically separate locations, especially to maintain data consistency. However, distributed protocols can leak confidential information unless carefully designed otherwise.

Distributed applications are often structured in terms of *transactions*, which are atomic groups of operations. For example, when ordering a book online, one or more transactions occur to ensure that the same book is not sold twice, and to ensure that the sale of a book and payment transfer happen atomically. Transactions are ubiquitous in modern distributed systems. Implementations include Google's Spanner [11], Postgres [29], and Microsoft's Azure Storage [9]. Common middleware such as Enterprise Java Beans [26] and Microsoft .NET [1] also support transactions.

Many such transactions are distributed, involving multiple autonomous participants (vendors, banks, etc.). Crucially, these participants may not be equally trusted with all data. Standards such as X/Open XA [2] aim specifically to facilitate transactions that span

multiple systems, but none address information leaks inherent to transaction scheduling.

Distributed transaction implementations are often based on the two-phase commit protocol (2PC) [17]. We show that 2PC can create unintentional channels through which private information may be leaked, and trusted information may be manipulated. We expect our results apply to other protocols as well.

There is a fundamental tension between providing strong consistency guarantees in an application and respecting the security requirements of the application's trust domains. This work deepens the understanding of this trade-off and demonstrates that providing both strong consistency and security guarantees, while not always possible, is not a lost cause.

Concretely, we make the following contributions in this paper:

- We describe *abort channels*, a new kind of side channel through which confidential information can be leaked in transactional systems (§2).

- We demonstrate exploitation of abort channels on a distributed system (§2.3).

- We define an abstract model of distributed systems, transactions, and information flow security (§3), and introduce *relaxed observational determinism*, a noninterference-based security model for distributed systems (§3.7.1).

- We establish that within this model, it is not possible for any protocol to securely serialize all sets of transactions, even if the transactions are individually secure (§4).

- We introduce and prove a sufficient condition for ensuring serializable transactions can be securely scheduled (§5).

- We define the *staged commit* protocol, a novel secure scheduling protocol for transactions meeting this condition (§6).

- We implement our novel protocol in the Fabric system [24], and extend the Fabric language and compiler to statically ensure transactions will be securely scheduled (§7).

- We evaluate the expressiveness of the new static checking discipline and the runtime overhead of the staged commit protocol (§8).

We discuss related work further in §9, and conclude in §10. For brevity, we present proof sketches of the results in this paper; full proofs can be found in the technical report [34].
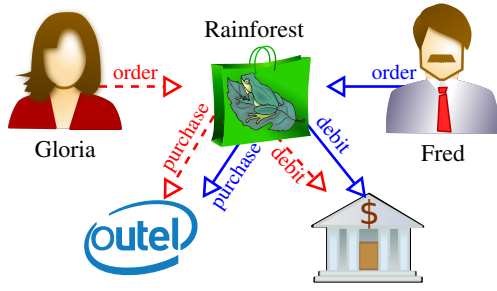
2. ABORT CHANNELS

Two transactions working with the same data can *conflict* if at least one of them is writing to the data. Typically, this means that one (or both) of the transactions has failed and must be *aborted*.

Figure 1: Rainforest example. Gloria and Fred each buy an Outel chip via Rainforest's store. Gloria's transaction is in red, dashed arrows; Fred's is in blue, solid arrows.

In many transaction protocols, including 2PC, a participant[1] involved in both transactions can abort a failed transaction by sending an *abort message* to all other participants in the failed transaction [17]. These abort messages can create unintended *abort channels*, through which private information can be leaked, and trusted information can be manipulated.

An abort message can convey secret information if a participant aborts a transaction otherwise likely to be scheduled, because another participant in the same transaction might deduce something about the aborting participant. For example, that other participant might guess that the abort is likely caused by the presence of another—possibly secret—conflicting transaction.

Conspirators might deliberately use abort channels to covertly transfer information within a system otherwise believed to be secure. Although abort channels communicate at most one bit per (attempted) transaction, they could be used as a high-bandwidth covert channel for exfiltration of sensitive information. Current transactional systems can schedule over 100 million transactions per second, even at modest system sizes [15]. It is difficult to know if abort channels are already being exploited in real systems, but large-scale, multi-user transactional systems such as Spanner [11] or Azure Storage [9] are in principle vulnerable.

Abort messages also affect the integrity of transaction scheduling. An abort typically causes a transaction not to be scheduled. Even if the system simply retries the transaction until it is scheduled, this still permits a participant to control the ordering of transactions, even if it has no authority to affect them. For example, a participant might gain some advantage by ensuring that its own transactions always happen after a competitor's.

Transactions can also create channels that leak information based on timing or termination [5, 8]. We treat timing and termination channels as outside the scope of this work, to be handled by mechanisms such as timing channel mitigation [22, 4, 7]. Abort channels differ from these previously identified channels in that information leaks via the existence of explicit messages, with no reliance on timing other than their ordering. Timing mitigation does not control abort channels.

2.1 Rainforest Example

A simple example illustrates how transaction aborts create a channel that can leak information. Consider a web-store application for the fictional on-line retailer Rainforest, illustrated in Fig. 1. Rainforest's business operates on data from suppliers, customers, and banks. Rainforest wants to ensure that it takes money from customers only if the items ordered have been shipped from the suppliers. As a result, Rainforest implements purchasing using serializable transactions. Customers expect that their activities do not influence each other, and that their financial information is not leaked

---

[1]Transaction participants are often processes or network nodes.
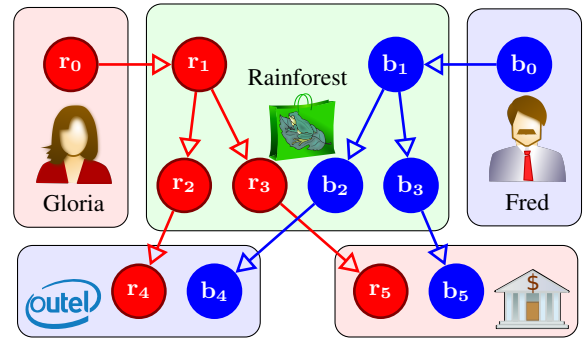
Figure 2: The events of the transactions in Fig. 1. Gloria's transaction consists of r_0, r_1, r_2, r_3, r_4, and r_5. Bob's consists of b_0, b_1, b_2, b_3, b_4, and b_5. Happens-before ($\rightarrow$) relationships are arrows. The shaded blocks around events indicate locations, and are labeled with participants from Fig. 1.

to suppliers. These expectations might be backed by law.

In Fig. 1, Gloria and Fred are both making purchases on Rainforest at roughly the same time. They each purchase an Outel chip, and pay using their accounts at CountriBank.

If Rainforest uses 2PC to perform both of these transactions, it is possible for Gloria to see an abort when Outel tries to schedule her transaction and Fred's. The abort leaks information about Fred's purchase at Outel to Gloria. Alternatively, if Gloria is simultaneously using her bank account in an unrelated purchase, scheduling conflicts at the bank might leak to Outel, which could thereby learn of Gloria's unrelated purchase.
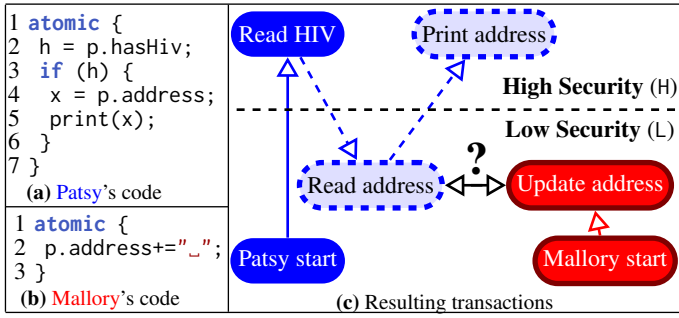
These concerns are about confidentiality, but transactions may also create integrity concerns. The bank might choose to abort transactions to affect the order in which Outel sells chips. Rainforest and Outel may not want the bank to have this power.

2.2 Hospital Example

As a second, running example, we use two small programs with an abort channel. Suppose Patsy is a trusted hospital employee, running the code in Fig. 3a to collect the addresses of HIV-positive patients in order to send treatment reminders. Patsy runs her transaction on her own computer, which she fully controls, but it interacts with a trusted hospital database on another machine. Patsy starts a transaction for each patient p, where transaction blocks are indicated by the keyword `atomic`. If p does not have HIV, the transaction finishes immediately. Fig. 3c shows the resulting transaction in solid blue. (Events in the transaction are represented as ovals; arrows represent dependencies between transaction events.) Otherwise, if the patient has HIV, Patsy's transaction reads the patient's address and prints it (the blue transaction in Fig. 3c, including dashed events).

Suppose Mallory is another employee at the same hospital, but is not trusted to know each patient's HIV status. Mallory is, however, trusted with patient addresses. Like Patsy, Mallory's code runs on her own computer, which she fully controls, but interacts with the trusted hospital database on another machine. She runs the code in Fig. 3b to update each patient's address in a separate transaction, resulting in the red transaction in Fig. 3c. When Mallory updates the address of an HIV-positive patient, her transaction might conflict with one of Patsy's, and Mallory would observe an abort. Thus Mallory can learn which patients are HIV-positive by updating each patient's address while Patsy is checking the patients' HIV statuses. Each time one of Mallory's transactions aborts, private information leaks: that patient has HIV.

One solution to this problem is to change Patsy's transaction: instead of reading the address only if the patient is HIV positive,

```
1 atomic {
2  h = p.hasHiv;
3  if (h) {
4    x = p.address;
5    print(x);
6  }
7 }
```
(a) Patsy's code

```
1 atomic {
2  p.address+=" ";
3 }
```
(b) Mallory's code

(c) Resulting transactions

Figure 3: Insecure hospital scenario. Patsy runs a program (3a) for each patient p. If p has HIV (which is private information), she prints out p's address for her records. The resulting transaction takes one of two forms. Both begin with the event Patsy start. If p is HIV negative, the transaction ends with Read HIV. Otherwise, it includes the blue events with dashed outlines. Meanwhile, Mallory updates the p's (less secret) address (3b), resulting in the transaction with red, solid-bordered events. This conflicts with Patsy's transaction, requiring the system to order the update and the read, exactly when p has HIV ("**?**" in 3c).

Patsy reads every patient's address. This illustrates a core goal of our work: identifying which programs can be scheduled securely. In Fig. 4a, lines 3 and 4 of Patsy's code have been switched. As Fig. 4c shows, both possible transactions read the patient's address. Since Mallory cannot distinguish which of Patsy's transactions has run, she cannot learn which patients have HIV.

2.3 Attack Demonstration

Using code resembling Fig. 3, we implemented the attack described in our hospital example (§ 2.2) using the Fabric distributed system [3, 24]. We ran nodes representing Patsy and Mallory, and a storage node for the patient records.

To improve the likelihood of Mallory conflicting with Patsy (and thereby receiving an abort), we had Patsy loop roughly once a second, continually reading the address of a single patient after verifying their HIV-positive status. Meanwhile, Mallory attempted to update the patient's address with approximately the same frequency as Patsy's transaction.
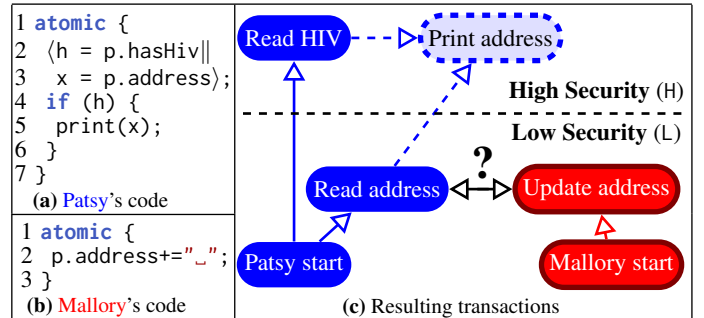
Like many other distributed transaction systems, Fabric uses two-phase commit. Mallory's window of opportunity for receiving an abort exists between the two phases of Patsy's commit, which ordinarily involves a network round trip. However, both nodes were run on a single computer. To model a cloud-based server, we simulated a 100 ms network delay between Patsy and the storage node.

We ran this experiment for 90 minutes. During this time, Mallory received an abort roughly once for every 20 transactions Patsy attempted. As a result, approximately every 20 seconds, Mallory learned that a patient had HIV. In principle, many such attacks could be run in parallel, so this should be seen as a minimal, rather than a maximal, rate of information leakage for this setup.

As described later, our modified Fabric compiler (§ 7) correctly rejects Patsy's code. We amended Patsy's code to reflect Fig. 4, and our implementation of the staged commit protocol (§ 6) was able to schedule the transactions without leaking information. Mallory was no more or less likely to receive aborts regardless of whether the patient had HIV.

3. SYSTEM MODEL

We introduce a formal, abstract system model that serves as our framework for developing protocols and proving their security properties. Despite its simplicity, the model captures the necessary fea-


```
1 atomic {
2  ⟨h = p.hasHiv∥
3   x = p.address⟩;
4  if (h) {
5    print(x);
6  }
7 }
```
(a) Patsy's code

```
1 atomic {
2  p.address+=" ";
3 }
```
(b) Mallory's code

(c) Resulting transactions

Figure 4: Secure hospital scenario. A secure version of Fig. 3, in which lines 3 and 4 of Patsy's code (3a) are switched, and the resulting lines 2 and 3 can be run in parallel ($\langle \parallel \rangle$). Thus the transaction reads p's address regardless of whether p has HIV, and so Mallory cannot distinguish which form Patsy's transaction takes.

tures of distributed transaction systems and protocols. As part of this model, we define what it means for transactions to be serializable and what it means for a protocol to serialize transactions both correctly and securely.

3.1 State and Events

Similarly to Lamport [23], we define a *system state* to include a finite set of *events*, representing a history of the system up to a moment in time. An event (denoted e) is an atomic native action that takes place at a *location*, which can be thought of as a physical computer on the network. Some events may represent read operations ("the variable x had the value 3"), or write operations ("2 was written into the variable y"). In Figures 3 and 4, for example, events are represented as ovals, and correspond to lines of code.

Also part of the system state is a causal ordering on events. Like Lamport's causality [23], the ordering describes when one event e_1 causes another event e_2. In this case, we say e_1 *happens before* e_2, written as $e_1 \rightarrow e_2$. This relationship would hold if, for example, e_1 is the sending of a message, and e_2 its receipt. The ordering ($\rightarrow$) is a strict partial order: irreflexive, asymmetric, and transitive. Therefore, $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ together imply $e_1 \rightarrow e_3$.

The arrows in Figures 2 to 4 show happens-before relationships for the transactions involved.

3.2 Information Flow Lattice

We extend Lamport's model by assigning to each event e a *security label*, written $\ell(e)$, which defines the confidentiality and integrity requirements of the event. Events are the most fine-grained unit of information in our model, so there is no distinction between the confidentiality of an event's *occurrence* and that of its *contents*. Labels in our model are similar to high and low event sets [30, 10]. In Figures 3 and 4, two security labels, High and Low (H and L for short), are represented by the events' positions relative to the dashed line.

For generality, we assume that labels are drawn from a lattice [12], depicted in Fig. 5. Information is only permitted to flow upward in the lattice. We write "$\ell(e_1)$ is below $\ell(e_2)$" as $\ell(e_1) \sqsubseteq \ell(e_2)$, meaning it is secure for the information in e_1 to flow to e_2.

For instance, in Fig. 3, information should not flow from any events labeled H to any labeled L. Intuitively, we don't want secret information to determine any non-secret events, because unauthorized parties might learn something secret. However, information can flow in the reverse direction: reading the patient's address (labeled L) can affect Patsy's printout (labeled H): L $\sqsubseteq$ H.

Like events, each location has a label, representing a limit on events with which that location can be trusted. No event should
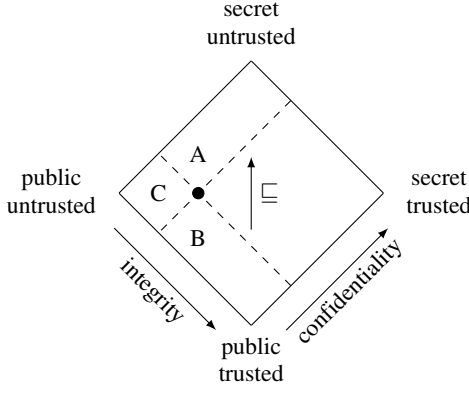
Figure 5: Security lattice: The dot represents a label in the lattice, and the dashed lines divide the lattice into four quadrants relative to this label. If the label represents an event, then only events with labels in quadrant B may influence this event, and this event may only influence events with labels in quadrant A. If the label represents a location, then only events with labels in quadrant C may occur at that location.

have more integrity than its location. Similarly, no event should be too secret for its location to know. Thus, in Fig. 5, only events *to the left of* a location's label (i.e., region C in the figure) may take place at that location.

For example, consider Gloria's payment event at CountriBank in the Rainforest example Fig. 1. This event (r_5 in Fig. 2) represents money moving from Gloria's account to Outel's. The label ℓ of r_5 should not have any more integrity than CountriBank itself, since the bank controls r_5. Likewise, the bank knows about r_5, so ℓ cannot be more confidential than the CountriBank's label. This would put ℓ to the *left* of the label representing CountriBank in the lattice of Fig. 5.

Our prototype implementation of secure transactions is built using the Fabric system [24], so the lattice used in the implementation is based on the Decentralized Label Model (DLM) [27]. However, the results of this paper are independent of the lattice used.

3.3 Conflicts

Two events in different transactions may *conflict*. This is a property inherent to some pairs of events. Intuitively, conflicting events are events that must be ordered for data to be consistent. For example, if e_1 represents reading variable x, and e_2 represents writing x, then they conflict, and furthermore, the value read and the value written establish an ordering between the events. Likewise, if two events both write variable x, they conflict, and the system must decide their ordering because it affects future reads of x.

In our hospital example (Figures 3 and 4), the events Read address and Update address conflict. Specifically, the value read will change depending on whether it is read before or after the update. Thus for any such pair of events, there is a happens-before ($\rightarrow$) ordering between them, in one direction or the other.

We assume that conflicting events have the same label. This assumption is intuitive in the case of events that are reads and writes to the same variable (that is, storage location). Read and write operations in separate transactions could have occurred in either order, so the happens-before relationship between the read and write events cannot be predicted in advance.

Our notion of *conflict* is meant to describe direct interaction between transactions. Hence, we also assume any conflicting events happen at the same location.
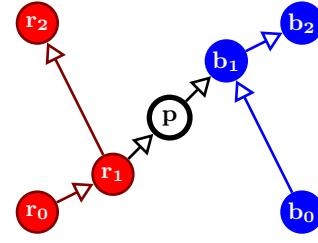
Figure 6: An example system state. The events $r_0, r_1,$ and r_2 form transaction **R**, and the events $b_0, b_1,$ and b_2 form transaction **B**. Event **p** is not part of either transaction. It may be an input, such as a network delay event, or part of a protocol used to schedule the transactions. In this state, $r_1 \rightarrow p \rightarrow b_1$, which means that r_1 happens before b_1, and so the transactions are ordered: $R \rightarrow B$.

3.4 Serializability and Secure Information Flow

Traditionally a transaction is modeled as a set of reads and writes to different objects [28]. We take a more abstract view, and model a transaction as a set of events that arise from running a piece of code. Each transaction features a *start event*, representing the decision to execute the transaction's code. Start events, by definition, happen before all others in the transaction. Multiple possible transactions can feature the same start event: the complete behavior of the transaction's code is not always determined when it starts executing, and may depend on past system events.

Fig. 4c shows two possible transactions, in blue, that can result from running the secure version of Patsy's code. They share the three events in solid blue, including the start event (Patsy start); one transaction contains a fourth event, Print address. The figure also shows in red the transaction resulting from Mallory's code. Fig. 6 is a more abstract example, in which r_0 is the start event of transaction **R**, and b_0 is the start event of transaction **B**.

In order to discuss what it means to *serialize* transactions, we need a notion of the *order* in which transactions happen. We obtain this ordering by lifting the happens-before relation on events to a happens-before ($\rightarrow$) relation for transactions. We say that transaction T_2 *directly depends* on T_1, written $T_1 \prec T_2$, if an event in T_1 happens before an event in T_2:

$$T_1 \prec T_2 \quad \equiv \quad T_1 \neq T_2 \land \exists e_1 \in T_1, e_2 \in T_2 \,.\, e_1 \rightarrow e_2$$

The happens-before relation on transactions ($\rightarrow$) is the transitive closure of this direct dependence relation $\prec$. Thus, in Fig. 6, the ordering $R \rightarrow B$ holds. Likewise, Fig. 7 is a system state featuring the transactions from our hospital example (Fig. 4), in which Patsy$\rightarrow$Mallory holds.

DEF. 1 (SERIALIZABILITY). *Transactions are serializable exactly when happens-before is a strict partial order on transactions.*

Any total order consistent with this strict partial order would then respect the happens-before ordering ($\rightarrow$) of events. Such a total ordering would represent a *serial order* of transactions.

DEF. 2 (SECURE INFORMATION FLOW). *A transaction is* information-flow secure *if happens-before* ($\rightarrow$) *relationships between transaction events—and therefore causality—are consistent with permitted information flow:*

$$e_1 \rightarrow e_2 \quad \implies \quad \ell(e_1) \sqsubseteq \ell(e_2)$$

This definition represents traditional information flow control within each transaction. Intuitively, each transaction itself cannot cause a security breach (although this definition says nothing about the protocol scheduling them). In our hospital example,
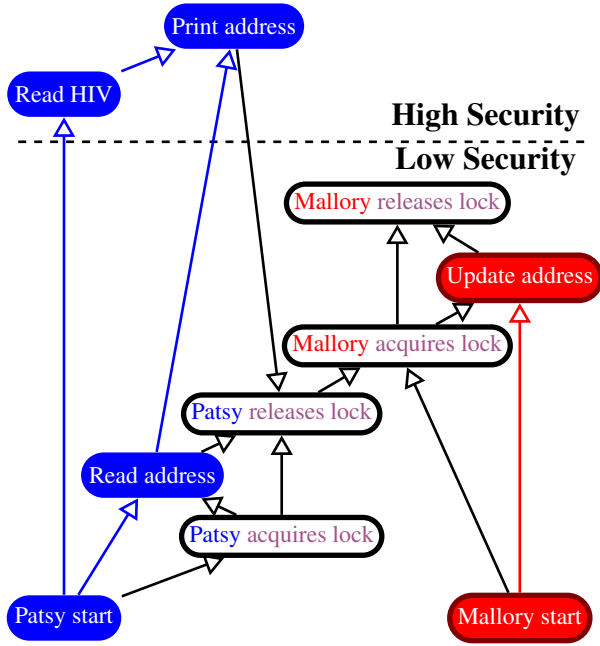
Figure 7: A possible system state after running transactions from Fig. 4c, assuming the patient has HIV, and an exclusive lock is used to order the transactions. (Events prior to everything in both transactions are not shown.) Because Patsy acquires the lock first, the transactions are ordered Patsy⇢Mallory. While each transaction is information-flow secure (a property of events within a transaction), when Patsy releases the lock after her transaction, a high security event happens before a low security one. We discuss secure scheduling protocols in §6.

Patsy's transaction in Fig. 3c is not *information-flow secure*, since Read HIV happens before Read address, and yet the label of Read HIV, H, does not flow to the label of Read address, L. However, in the modified, secure version (Fig. 4c), there are no such insecure happens-before relationships, so Patsy's transaction is secure.

3.5 Network and Timing

Although this model abstracts over networks and messaging, we consider a message to comprise both a *send event* and a *receive event*. We assume asynchronous messaging: no guarantees can be made about network delay. Perhaps because this popular assumption is so daunting, many security researchers ignore timing-based attacks. There are methods for mitigating leakage via timing channels [22, 4, 7] but in this work we too ignore timing.

To model nondeterministic message delay, we introduce a *network delay event* for each message receipt event, with the same label and location. The network delay event may occur at any time after the message send event. It must happen before ($\rightarrow$) the corresponding receipt event. In Fig. 6, event r_1 could represent sending a message, event p could be the corresponding network delay event, which is not part of any transaction, and event b_1 could be the message receipt event. Fig. 6 does not require p to be a network delay event. It could be any event that is not part of either transaction. For example, it might be part of some scheduling protocol.

3.6 Executions, Protocols, and Inputs

An *execution* is a start state paired with a totally ordered sequence of events that occur after the start state. This sequence must be consistent with happens-before ($\rightarrow$). Recall that a system state is a set of events (§3.1). Each event in the sequence therefore corresponds to a system state containing all the events in the start state,

and all events up to and including this event in the sequence. Viewing an execution as a sequence of system states, an event is *scheduled* if it is in a state, and once it is scheduled, it will be scheduled in all later states. Two executions are *equivalent* if their start states are equal, and their sequences contain the same set of events, so they finish with equal system states (same set of events, same $\rightarrow$). A *full execution* represents the entire lifetime of the system, so its start state contains no events.

For example, Fig. 8 illustrates two equivalent full executions ending in the system state from Fig. 6.

A transaction scheduling *protocol* determines the order in which each location schedules the events of transactions. Given a set of possible transactions, a location, and a set of events representing a system state at that location, a protocol decides which event is scheduled next by the location:

$$protocol : \mathbf{set} \langle \text{Transactions} \rangle \times \text{Location} \times \text{State} \rightarrow \text{event}$$

Protocols can schedule an event from a started (but unfinished) transaction, or other events used by the protocol itself. In order to schedule transaction events in ways that satisfy certain constraints, like serializability, protocols may have to schedule additional events, which are not part of any transaction. These can include message send and receipt events. For example, in Fig. 7, the locking events are not part of any transaction, but are scheduled by the protocol in order to ensure serializability.

Certain kinds of events are not scheduled by protocols, because they are not under the control of the system. Events representing external inputs, including the start events of transactions, can happen at any time: they are fundamentally nondeterministic. We also treat the receive times of messages as external inputs. Each message receive event is the deterministic result of its send event and of a nondeterministic *network delay event* featuring the same security label as the receive event. We refer to start and network delay events collectively as *nondeterministic input events* (NIEs).

Protocols do not output NIEs. Instead, an NIE may appear at any point in an execution, and any prior events in the execution can happen before ($\rightarrow$) the NIE. Recall that an execution features a sequence of events, each of which can be seen as a system state featuring all events up to that point. An execution E is consistent with a protocol p if every event in the sequence is either an NIE, or the result of p applied to the previous state at the event's location. We sometimes say p *results in* E to mean "E is consistent with p."
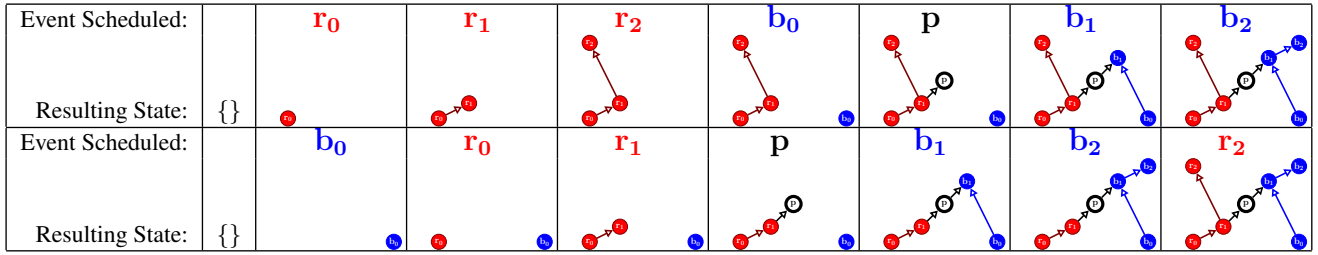
As an example, assume all events in Fig. 6 have the same location L, and no messages are involved. Start events r_0 and b_0 are NIEs. Every other event has been scheduled by a protocol. Fig. 8 shows two different executions, which may be using different protocols, determining which events to schedule in each state. We can see that in the top execution of Fig. 8, the protocol maps:

$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0\} \mapsto r_1$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, r_1\} \mapsto r_2$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, r_1, r_2, b_0\} \mapsto p$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, r_1, r_2, b_0, p\} \mapsto b_1$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, r_1, r_2, b_0, p, b_1\} \mapsto b_2$$

The protocol in the bottom execution of Fig. 8 maps:

$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, b_0\} \mapsto r_1$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, b_0, r_1\} \mapsto p$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, b_0, r_1, p\} \mapsto b_1$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, b_0, r_1, p, b_1\} \mapsto b_2$$
$$\{\mathbf{R}, \mathbf{B}, \ldots\}, L, \{r_0, b_0, r_1, p, b_1, b_2\} \mapsto r_2$$

Ultimately, a protocol must determine the ordering of transactions. If the exact set of start events to be scheduled (as opposed

233

Event Scheduled:		r_0	r_1	r_2	b_0	p	b_1	b_2
Resulting State:	{}							
Event Scheduled:		b_0	r_0	r_1	p	b_1	b_2	r_2
Resulting State:	{}							

Figure 8: Two equivalent full executions for the system state from Fig. 6. Each begins with a start state (the empty set for full executions), followed by a sequence of events, each of which corresponds to the resulting system state.

to start events possible) were always known in advance, scheduling would be trivial. A protocol should not require one transaction to run before another *a priori*: **start events from any subset of possible transactions may be scheduled at any time.** No protocol should result in a system state in which such a start event cannot be scheduled, or an incomplete transaction can never finish.

3.7 Semantic Security Properties

Consider an observer who can only "see" events at some security level ℓ or below. If two states S_1 and S_2 are indistinguishable to the observer, then after a program runs, noninterference requires that the resulting executions remain indistinguishable to the observer. Secret values, which the observer cannot see, may differ in S_1 and S_2, and may result in different states at the end of the executions, but the observer should not be able to see these differences.

3.7.1 Relaxed Observational Determinism

Semantic conditions for information security are typically based on some variant of noninterference [19, 31]. These variants are often distinguished by their approaches to nondeterminism. However, many of these semantic security conditions fail under *refinement*: if some nondeterministic choices are fixed, security is violated [37]. However, low-security observational determinism [30, 37] is a strong property that is secure under refinement: intuitively, if an observer with label ℓ cannot distinguish states S and S', that observer must not be able to distinguish any execution E beginning with S from *any* execution E' beginning with S':

$$\left(S \approx_\ell S'\right) \Rightarrow E \approx_\ell E'$$

This property is *too* strong because it rules out two sources of nondeterminism that we want to allow: first, the ability of any transaction to start at any time, and second, network delays. Therefore, we relax observational determinism to permit certain nondeterminism. We only require that executions be indistinguishable to the observer if their NIEs are indistinguishable to the observer:

$$\left(S \approx_\ell S' \quad \wedge \quad NIE(E) \approx_\ell NIE\left(E'\right)\right) \Rightarrow E \approx_\ell E'$$

We call this relaxed property *relaxed observational determinism*. It might appear to be equivalent to observational determinism, but with the NIEs encoded in the start states. This is not the case. If NIEs were encoded in the start states, protocols would be able to read which transactions will start and when messages will arrive in the future. Therefore relaxed observational determinism captures something that observational determinism does not: unknowable but "allowed" nondeterminism at any point in an execution.

By deliberately classifying start events and network delays as input, we allow certain kinds of information leaks that observational determinism would not. Specifically, a malicious network could leak information by manipulating the order or timing of message delivery. However, such a network could by definition communicate information to its co-conspirators anyway. Information can also be leaked through the order or timing of start events. This problem is beyond the scope of this work.

Conditioning the premise of the security condition on the indistinguishability of information that is allowed to be released is an idea that has been used earlier [32], but not in this way, to our knowledge.

In our hospital example, as illustrated in Fig. 4, the system determines which of Patsy's transactions (the one with the dashed events, or the one without the dashed events) will run based on whether p.hasHiv is true. We can consider p.hasHiv's value to be a high-security event that happens before all reads of p.hasHiv. If we classify this past high-security event as input, and all low-security events as low-observable for Mallory, then we must ensure that when Patsy's code runs, the low-security projections of resulting executions are always the same, regardless of whether p.hasHiv. Patsy's possible transactions in Fig. 4 allow for observational determinism, while her transactions in Fig. 3 do not, since whether or not Read address occurs depends on p.hasHiv. Whether or not the system actually maintains observational determinism, however, depends on the protocol scheduling the events.

DEF. 3 (PROTOCOL SECURITY). *A protocol is considered* secure *if the set of resulting executions satisfies relaxed observational determinism for any allowed sets of information-flow secure transactions and any possible NIEs.*

4. IMPOSSIBILITY

One of our contributions is to show that even in the absence of timing channels, there is a fundamental conflict between secure noninterference and serializability. Previous results showing such a conflict, for example the work of Smith et al. [36] consider only confidentiality and show only that timing channels are unavoidable.

THEOREM 1 (IMPOSSIBILITY). *No secure protocol[2] can serialize all possible sets of information-flow secure transactions.[3]*

We assume protocols cannot simply introduce an arbitrarily trusted third party; a protocol must be able to run using only the set of locations that have events being scheduled.

PROOF SKETCH. Consider the counterexample shown in Fig. 9. Alice and Bob are both cloud computing providers who keep strict logs of the order in which various jobs start and stop. Highly trusted (possibly government) auditors may review these logs, and check for consistency, to ensure cloud providers are honest and fair. As

---

[2]barring unforeseen cryptographic capabilities (§4.1)

[3]In fact, what we prove is stronger. Our proof holds for even possibilistic security conditions [25], which are weaker than relaxed observational determinism (see technical report [34]). No protocol whose resulting traces satisfy even this weaker condition can serialize all sets of information-flow secure transactions.
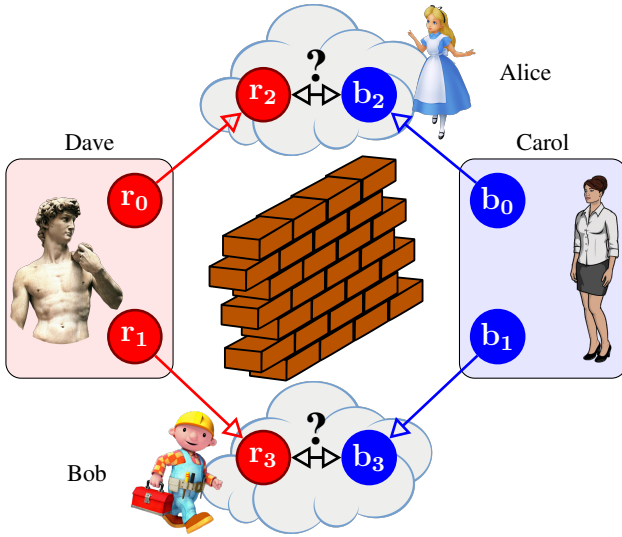
Figure 9: Transactions that cannot be securely serialized. Dave's transaction includes r_0, r_1, r_2, and r_3. Carol's includes b_0, b_1, b_2, and b_3. Cloud providers Alice and Bob must decide how to order their events. Alice and Bob may not influence each other, and Carol and Dave may not influence each other, as represented by the wall. For these transactions to be serializable, Alice's ordering of r_2 and b_2 must agree with Bob's ordering of r_3 and b_3.

competitors, Alice and Bob do not want each other to gain any information about their services, and do not trust each other to affect their own services.

Carol and Dave are presently running jobs on Alice's cloud. Both Carol and Dave would like to stop their jobs on Alice's cloud, and start new ones on Bob's cloud. Each wants to do this atomically, effectively maintaining exactly one running job at all times. Carol and Dave consider their jobs to be somewhat confidential; they do not want each other to know about them. Unlike the example from Fig. 1, Dave and Carol's transactions do not go through a third party like Rainforest. For the transactions to be serializable, Alice's ordering of the old jobs stopping must agree with Bob's ordering of the new jobs starting.

In any system with an asynchronous network, it is possible to reach a state in which Carol's message to Alice has arrived, but not her message to Bob, and Dave's message to Bob has arrived, but not his message to Alice. In this state, neither Alice nor Bob can know whether one or both transactions have begun. It is impossible for either to communicate this information to the other without violating relaxed observational determinism. Specifically, any protocol that relayed such information from one cloud provider to the other would allow the recipient to distinguish the order of message delivery to the other cloud provider. That ordering is considered secret input, and so this would be a security violation. All executions with identical start states, and identical inputs visible to Alice, but differently ordered network delay events at Bob, which are inputs invisible to Alice, would become distinguishable to Alice. □

4.1 Cryptography

This essentially information-theoretic argument does not account for the possibility that some protocol could produce *computationally indistinguishable* traces that are low-distinguishable with sufficient computational power (e.g., to break encryption). However, we are unaware of any cryptographic protocols that would permit Alice and Bob to learn a consistent order in which to schedule events without learning each other's confidential information.

5. ANALYSIS

Although secure scheduling is impossible in general, many sets of transactions can be scheduled securely. We therefore investigate which conditions are sufficient for secure scheduling, and what protocols can function securely under these conditions.

5.1 Monotonicity

A relatively simple condition suffices to guarantee schedulability, while preserving relaxed observational determinism:

DEF. 4 (MONOTONICITY). *A transaction is* monotonic *if it is information-flow secure and its events are totally ordered by happens-before* $(\twoheadrightarrow)$.

THEOREM 2 (MONOTONICITY $\Rightarrow$ SCHEDULABILITY).
A protocol exists that can serialize any set of monotonic transactions and preserve relaxed observational determinism.

PROOF SKETCH. Monotonicity requires that each event must be allowed to influence all future events in the transaction. A simple, pessimistic transaction protocol can schedule such transactions securely. In order to define this protocol, we need a notion of *locks* within our model.

Locks. A lock consists of an infinite set of events for each allowed transaction. A transaction *acquires* a lock by scheduling any event from this set. It *releases* a lock by scheduling another event from this set. Thus, in a system state S, a transaction T *holds* a lock if S contains an odd number of events from the lock's set corresponding to T. No correct protocol should result in a state in which multiple transactions hold the same lock. All pairs of events in a lock conflict, so scheduled events that are part of the same lock must be totally ordered by happens-before $(\twoheadrightarrow)$. All events in a lock share a location, which is considered to be the location of the lock itself. Likewise, all events in a lock share a label, which is considered to be the label of the lock itself.

A critical property for transaction scheduling is *deadlock freedom* [17, 35], which requires that a protocol can eventually schedule all events from any transaction whose start event has been scheduled. A system enters *deadlock* when it reaches a state after which this is not the case. For example, deadlock happens if a protocol requires two transactions each to wait until the other completes: both will wait forever. If all transactions are finite sets of events (i.e., all transactions can terminate), then deadlock freedom guarantees that a system with a finite set of start events eventually terminates, a liveness property.

We now describe a deadlock-free protocol that can securely serialize any set of monotonic transactions, and preserve relaxed observational determinism:

- Each event in each transaction has a corresponding lock, except start events.

- Any events that have the same label share a lock, and this lock shares a location with at least one of the events. Conflicting events are assumed to share a label (§ 3.4).

- A transaction must hold an event's lock to schedule that event.

- A transaction acquires locks in sequence, scheduling events as it goes. Since all events are ordered according to a global security lattice, all transactions that acquire the same locks do so in the same order. Therefore they do not deadlock.

- If a lock is already held, the transaction waits for it to be released.

- When all events are scheduled, the transaction commits, releasing locks in reverse order. Any messages sent as part of the transaction would thus receive a reply, indicating only that the message had been received, and all its repercussions committed. We call these replies *commit* messages.

- For each location, the protocol rotates between all uncommitted transactions, scheduling any intermediate events (such as lock acquisitions) until it either can schedule one event in the transaction or can make no progress, and then rotates to the next transaction.

Security Intuition. Acquiring locks shared by multiple events on different locations requires a commit protocol between those locations. However, this does not leak information because all locations involved are explicitly allowed to observe and influence all events involved. Therefore several known commit protocols will do, including 2PC. Since the only messages sent as part of the protocol are commit messages, and each recipient knows it will receive a commit message by virtue of sending a message in the protocol, no information (other than timing) is transferred by the scheduling mechanism itself. □

5.2 Relaxed Monotonicity

Monotonicity, while relatively easy to understand, is not the weakest condition we know to be sufficient for secure schedulability. It can be substantially relaxed. In order to explain our weaker condition, *relaxed monotonicity*, we first need to introduce a concept we call *visibility*:

DEF. 5 (VISIBLE-TO). *An event e in transaction T is* visible to *a location L if and only if it happens at L, or if there exists another event $e' \in T$ at L, such that $e \rightarrow e'$.*

DEF. 6 (RELAXED MONOTONICITY). *A transaction T satisfies* relaxed monotonicity *if it is information-flow secure and for each location L, all events in T visible to L happen before all events in T not visible to L.*

In §6, we demonstrate that relaxed monotonicity guarantees schedulability. Specifically, we present a staged commit protocol, and prove that it schedules any set of transactions satisfying relaxed monotonicity, while preserving relaxed observational determinism (Thm. 4).

5.3 Requirements for Secure Atomicity

Monotonicity and relaxed monotonicity are sufficient conditions for a set of transactions to be securely schedulable. Some sets of transactions meet neither condition, but can be securely serialized by some protocol. For example, any set of transactions that each happen entirely at one location can be securely serialized if each location schedules each transaction completely before beginning the next. We now describe a relatively simple condition that is necessary for any set of transactions to be securely scheduled.

Decision Events and Conflicting Events

In order to understand this necessary condition, we first describe *decision events* and *conflicting events*.

Borrowing some terminology from Fischer, Lynch, and Paterson [18], for a pair of transactions T_1 and T_2, any system state is either *bivalent* or *univalent*. A system state is *bivalent* with respect to T_1 and T_2 if there exist two valid executions that both include that state, but end with opposite orderings of T_1 and T_2. A system state is *univalent* with respect to T_1 and T_2 otherwise: for one

ordering of the transactions, no valid execution ending with that ordering contains the state.

We can define a similar relationship for start events: for any pair of distinct start events s_1 and s_2, a system state is bivalent with respect to those events if it features in two valid executions, both of which have s_1 and s_2 in scheduled transactions, but those transactions are in opposite order. A system state is univalent with respect to s_1 and s_2 otherwise.

All full executions (i.e., those starting with an empty state) that order a pair of transactions begin in a *bivalent* state with respect to their start events, before either is scheduled. By our definition of serializability and transaction ordering, once transactions are ordered, they cannot be un-ordered. Any execution that orders the transactions therefore ends in a *univalent* state with respect to their start events. Any such execution consists of a sequence of 0 or more *bivalent* states followed by a sequence of *univalent* states. The event that is scheduled in the first *univalent* state, in a sense, decides the ordering of the transactions. We call it the *decision event*.

We call any event in T_1 or T_2 that conflicts with an event in the other transaction a *conflicting event*.

LEMMA 1 (DECISION EVENT → CONFLICTING EVENTS). *For any univalent state S with $T_1 \rightarrow T_2$, there exists a full execution E ending in S featuring a decision event e_d that happens before ($\rightarrow$) all conflicting events in T_1 and T_2 (other than e_d itself, if e_d is a conflicting event).*

PROOF SKETCH. We show that the contradiction implies an infinite chain of equivalent executions with earlier and earlier non-decision conflicting events, which is impossible given that system states are finite. □

We show that two fundamental system state properties are necessary for secure scheduling:

DEF. 7 (FIRST-PRECEDES-DECISION). *State S satisfies* First-Precedes-Decision *if, for any pair of transactions T_1 and T_2 in S with $T_1 \rightarrow T_2$, there is a full execution E ending in S with a decision event e_d that either is in T_1, or happens after an event in T_1.*

DEF. 8 (DECISION-PRECEDES-SECOND). *A state S satisfies* Decision-Precedes-Second *if, for any pair of transactions T_1 and T_2 in S with $T_1 \rightarrow T_2$, there is a full execution E' ending in S with a decision event e'_d, such that no event in T_2 happens before e'_d.*

Therefore, for a protocol to be secure, it must ensure resulting system states have these properties.

THEOREM 3 (NECESSARY CONDITION). *Any secure, deadlock-free protocol p must ensure that all full executions consistent with p feature only states satisfying both First-Precedes-Decision and Decision-Precedes-Second.*

PROOF. Given $T_1 \rightarrow T_2$, any execution E' ending in S features a decision event e_d. Decision events for the same pair of transactions in equivalent executions must agree on ordering, by the definition of equivalent execution. If T_1 does not contain E's decision event, e_d, or any event that happens before e_d, then there exists an equivalent execution in which e_d is scheduled before any events in T_1 or T_2. This execution would imply the existence of a system state in which no event in either transaction is scheduled, but it is impossible to schedule T_2 before T_1, regardless of inputs after that state. If, after this state, the start event for T_2 were scheduled, but not the start event for T_1, then T_2 cannot be scheduled. This contradicts a the deadlock-freedom requirement: no protocol should

result in a system state in which a supported transaction can never be scheduled.

Therefore some event in T_1 either is or happens before e_d for some full execution E ending in S.

If T_1 and T_2 conflict, then e'_d either is an event in T_1 or happens before an event in T_1, by Lemma 1. If an event $e_2 \in T_2$ happens before e'_d, then either $e'_d \in T_1$, and

$$e_2 \twoheadrightarrow e'_d \Rightarrow T_2 \twoheadrightarrow T_1$$

which is impossible, by the definition of happens-before, or $\exists e_1 \in T_1 . e'_d \twoheadrightarrow e_1$, and

$$e_2 \twoheadrightarrow e'_d \twoheadrightarrow e_1 \Rightarrow e_2 \twoheadrightarrow e_1 \Rightarrow T_2 \twoheadrightarrow T_1$$

which is also impossible, by the definition of happens-before.

If T_1 and T_2 do not conflict, then the only way $T_1 \twoheadrightarrow T_2$ implies that there exists some chain $T_1 \twoheadrightarrow T_3 \twoheadrightarrow T_4 \twoheadrightarrow \ldots \twoheadrightarrow T_n \twoheadrightarrow T_2$ such that and each transaction in the chain conflicts with the next. Therefore, by the above proof, an equivalent execution exists in which each transaction in the chain contains the decision event for ordering itself and the following transaction, and no events in the following transaction are before that decision event.

Therefore there exists some equivalent execution E' in which no event in T_2 happens before the decision event e'_d deciding the ordering between T_1 and T_2. $\quad\square$

Although Thm. 3 may seem trivial, it represents some important conclusions: No protocol can make any final ordering decision until at least one transaction involved has begun. Furthermore, it is impossible for the later transaction to determine the decision. Truly atomic transactions cannot include any kind of two-way interaction or negotiation for scheduling.

6. THE STAGED COMMIT PROTOCOL

We now present the staged commit protocol (SC) and prove that it is secure, given transactions satisfying relaxed monotonicity.

SC is a hybrid of traditional serialization protocols, such as 2PC, and the simple pessimistic protocol described in the proof of Thm. 2. Compared to our simple pessimistic protocol, it allows a broader variety of transactions to be scheduled (relaxed monotonicity vs. regular monotonicity), which in turn allows more concurrency. A transaction is divided into *stages*, each of which can be securely committed using a more traditional protocol. The stages themselves are executed in a pessimistic sequence.

Each event scheduled is considered to be either *precommitted* or *committed*. We express this in our model by the presence or absence of an "isCommitted" event corresponding to every event in a transaction. Intuitively, a *precommitted* event is part of some ongoing transaction, so no conflicting events that happen after a precommitted event should be scheduled. A *committed* event, on the other hand, is part of a completed transaction; conflicting events that happen after a committed event can safely be scheduled. Once an event is precommitted, it can never be un-scheduled. It can only change to being committed. Once an event is committed, it can never change back to being precommitted.

- The events of each transaction are divided into stages. Each stage will be scheduled using traditional 2PC, so aborts within a stage will be sent to all locations involved in that stage.

  To divide the events into stages, we establish equivalence classes of the events' labels. Labels within each class are *equivalent* in the following sense: when events with equivalent labels are aborted, those aborts can securely flow to the same set of locations. An event's abort can always flow to the

  event's own location, so locations involved in a stage can securely ensure the atomicity of the events in that stage. Since conflicting events have the same security labels, they will be in the same equivalence class. We call these equivalence classes *conflict labels* (cl).

- Each stage features events of the same conflict label, and is scheduled with 2PC. One location must coordinate the 2PC. All potential aborts in the stage must flow to the coordinator, and some events on the coordinator must be permitted to affect all events in the stage. Relaxed monotonicity implies that at least one such location exists for each conflict label.

  When a stage tries to schedule an event, but finds a precommitted conflicting event, it aborts the entire stage. Because conflicting events have the same label, these aborts cannot affect events on unpermitted locations.

  When a stage's 2PC completes, the events in the stage are scheduled, and considered *precommitted*.

- Each transaction precommits its stages as they occur. To avoid deadlock, we must ensure that whenever two transactions feature stages with equal conflict labels, they precommit those stages in the same order. Therefore, the staged commit protocol assumes an ordering of conflict labels. This can be any arbitrary ordering, so long as (1) it totally orders the conflict labels appearing in each transaction, and (2) all transactions agree on the ordering.

- When all stages are precommitted, all events in the transaction can be committed. *Commit* messages to this effect are sent between locations, backwards through the stages. Whenever an event in one stage triggers an event in the next, the locations involved can be sure a *commit* message will take the reverse path. The only information conveyed is timing.

Because events in a precommitted stage cannot be un-scheduled or "rolled back", a participant that is involved only in an earlier stage is prevented from gleaning any information about later stages. The participant will only learn, eventually, that it can commit. Patsy's transaction in Fig. 4c has at least two stages when the patient has HIV:

1. Patsy begins the transaction (Patsy start), and reads the address (Read Address). This stage will be atomically precommitted, and this precommit process will determine the relative ordering of Patsy's transaction and Mallory's, independent of more secret events.

2. Patsy finds that the patient has HIV (Read HIV), and prints the patient's address (Print address).

THEOREM 4 (SECURITY OF SC). *Any set of transactions satisfying relaxed monotonicity are serialized by SC securely without deadlock.*

PROOF SKETCH. **Security.** SC preserves relaxed observational determinism. Intuitively, any information flows that it adds are already included in the transaction.

SC adds no communication affecting security:

- Communication within each stage is strictly about events that all participants can both observe.

- For each pair of consecutive stages, at least one participant from the first stage can notify a participant in the second

stage securely, when it is time for the second stage to begin. Relaxed monotonicity ensures the second stage contains an event that happens after an event in the first stage, representing a line of communication.

- Communication for commits can safely proceed in reverse order of stages. Each participant knows when it precommits exactly which commit messages it will receive.

Serializability. Our proof is built around the following lemma: any execution in which an event in a transaction is committed features a system state in which all events in the transaction are precommitted. This lemma is used to show that SC guarantees a strict partial order of transactions, and therefore serializability.

Deadlock Freedom. Deadlock cannot form within any stage, since stages use 2PC, which preserves deadlock freedom. The stages themselves, like locks in our proof of Thm. 2, are precommitted in a consistent order, guaranteeing deadlock freedom. $\square$

The Importance of Optimism

SC specifies only a commit protocol. Actual computation (which generates the set of events) for each transaction can be done in advance, optimistically. If one stage precommits and the next is blocked by a conflicting transaction, optimistically precomputed events would have to be *rolled back*. However, no precommitted event need be rolled back. In fact, it would be insecure to do so. Thus SC allows for partially optimistic transactions with partial rollback.

Our model requires only that a transaction be a set of events. In many cases, however, it is not possible to know which transaction will run when a start event is scheduled. For example, a transaction might read a customer's banking information from a database and contact the appropriate bank. It would not be possible to know which bank should have an event in the transaction beforehand. If a system attempted to read the banking information prior to the transaction, then serializability is lost: the customer might change banks in between the read and the transaction, and so one might contact the wrong bank.

Optimism solves this problem: events are precomputed, and when an entire stage is completed, that stage's 2PC begins. This means that optimism is not just an optimization; it is required for secure scheduling in cases where the transactions' events are not known in advance.

7. IMPLEMENTATION

We extended the Fabric language and compiler to check that transactions can be securely scheduled, and we extended the Fabric runtime system to use SC. Fabric and IFDB [33] are the two open-source systems we are aware of that support distributed transactions on persistent, labeled data with information flow control. Of these, we chose Fabric for its static reasoning capabilities. IFDB checks labels entirely dynamically, so it cannot tell if a transaction is schedulable until after it has begun.

7.1 The Fabric Language

The Fabric language is designed for writing distributed programs using atomic transactions that operate on persistent, Java-like objects [24]. It has types that label each object field with information flow policies for confidentiality and integrity. The compiler uses these labels to check that Fabric programs enforce a noninterference property. However, like all modern systems built using 2PC, Fabric does not require that transactions be securely scheduled according to the policies in the program. Consequently, until now, abort channels have existed in Fabric.

```
1 atomic {                          PC  Possible conflictors
2   String{ℓ} p = post.read();    ⊥   {Alice, Bob, Carol}
3   Comments{ℓ'} c;                ⊥    -
4   if (p.contains("fizz")) {     ⊥    -
5     c.write("buzz");             ℓ   {Alice, Carol}
6   if (p.contains("buzz")) {     ⊥    -
7     c.write("fizz");             ℓ   {Alice, Carol}
8   }
9 }
```

Figure 10: Carol's program in our Blog example: Carol reads a post with label ℓ, and depending on what she reads, writes a comment with label ℓ'. Label ℓ permits Alice, Bob, and Carol to read the post, while ℓ' keeps the Comments more private and allows only Alice and Carol to view or edit.

We leverage these security labels and extend the compiler to additionally check that transactions in a Fabric program are monotonic (§ 5). This implementation prevents confidentiality breaches via abort channels. Preventing integrity breaches would require further dynamic checks, which we leave to future work.

7.2 Checking Monotonicity

Our modification to the Fabric compiler enforces relaxed monotonicity (Def. 6). Our evaluation (§ 8) shows that enforcing this condition does not exclude realistic and desirable programs. Our changes to the Fabric compiler and related files include 4.1k lines of code (out of roughly 59k lines).

7.2.1 Events and Conflict Labels in Fabric

The events in the system model (§ 3) are represented in our implementation by read and writes on fields of persistent Fabric objects. The label of the field being read or written corresponds to the event labels in our model.

SC (§ 6) divides events into stages based on conflict labels (`cl`). In our implementation, we define the `cl` of an event e to correspond to the set of *principals* authorized to read or write the field that is being accessed by e. If e is a write event, this set contains exactly those principals that can perform a conflicting operation (and thereby receive an abort); if e is a read event, the set is a conservative over-approximation, since only the writers can conflict.

Fig. 10 presents a program in which Carol schedules two events within a single transaction. First, she reads a blog post with security label ℓ. Second, she writes a comment (whose content depends on that of the post) with label ℓ'. Since ℓ permits Alice, Bob, or Carol to read the post, the `cl` of the first event includes all three principals. However, only Alice and Carol can read or write the comment, so when Carol goes to write it, only Alice or another transaction acting on behalf of Carol could cause conflicts. The `cl` of the write therefore includes only Alice and Carol.

7.2.2 Program Counter Label

The *program counter* label (`pc`) [13] labels the program context. For any given point in the code, the `pc` represents the join (least upper bound) of the labels of events that determine whether or not execution reaches that point in the code. These events include those occurring in if-statement and loop conditionals. For instance, in Fig. 10, whether line 5 runs depends on the value of p, which has label ℓ. Therefore, the fact that line 5 is executing is as secret as p, and the `pc` at line 5 is ℓ.

SC requires that when events with the same `cl` are aborted, those aborts can securely flow to the same set of locations. When an event causes an abort, the resulting abort messages carry information about the context in which the event occurs. Therefore, we enforce the requirement by introducing a constraint on the program

context in which events may occur: the pc must flow to the principals in the conflict label.

$$pc \sqsubseteq cl \qquad (1)$$

Eliding the details of how Fabric's labels are structured, in Fig. 10, $\bot$ flows to everything, and ℓ, the label of the blog post, does flow to the conflict label, indicating that both Alice and Carol can cause a conflict. Therefore, Eqn. (1) holds on lines 2, 5, and 7.

7.2.3 Ordering Stages

Each stage consists of operations with the same cl. To ensure all transactions precommit conflicting stages in the same order, we adopt a universal stage ordering:

$$principals(cl_i) \supsetneq principals(cl_{i+1}) \qquad (2)$$

The set of principals in each stage must be a strict superset of the principals in the next one. This ensures that unrestricted information can be read in one stage and sensitive information can be modified in a later stage in the same transaction. In the hospital example (Fig. 4), Read HIV has a conflict label that only includes trusted personnel, while Read address has a conflict label that includes more hospital staff. As a result, our implementation requires that Read address be staged before Read HIV in Patsy's transaction.

In Fig. 10, our stage ordering means that the read on line 2, with a cl of $\{Alice, Bob, Carol\}$ belongs in an earlier stage than the write, which features a cl of only $\{Alice, Carol\}$.

7.2.4 Method Annotations

To ensure modular program analysis and compilation, each method is analyzed independently. Fabric is an object-oriented language with dynamic dispatch, so it is not always possible to know in advance which method implementation a program will execute. Therefore, the exact conflict labels for events within a method call are not known at compile time. In order to ensure each atomic program can divide into monotonic stages, we annotate each method with bounds on the conflict labels of operations within the method. These annotations are the security analogue of argument and return types for methods.

7.3 Implementing SC

We extended the Fabric runtime system to use SC instead of traditional 2PC, modifying 2.4k lines of code out of a total of 24k lines of code in the original implementation. Specifically, we changed Fabric's 2PC-based transaction protocol so that it leaves each stage prepared until all stages are ready, and then commits.

Since Fabric labels can be dynamic, the compiler statically determines *potential stagepoints*—points in the program that may begin a new stage—along with the conflict labels of the stages immediately surrounding the potential stagepoint. If the compiler cannot statically determine whether the conflict labels before and after a stagepoint will be different, it inserts a dynamic equivalence check for the two labels. At run time, if the two labels are not equivalent, then a stage is ending, and the system precommits all operations made thus far. To precommit a stage, we run the first ("prepare") phase of 2PC. If there is an abort, the stage is re-executed until it eventually precommits.

In Fig. 10, there is a potential stagepoint before lines 4 and 6, where the next operation in each case will not include Bob as a possible conflictor. The conflict labels surrounding the potential stagepoint are $\{Alice, Bob, Carol\}$ (from reading the post on line 2) and $\{Alice, Carol\}$ (from writing the comment on either line 4 or 6). If another transaction caused the first stage to abort, then

Data item	Readers	Writers
Gloria's account balance	Bank, Gloria	Bank
Item price	(public)	Outel
Inventory	Outel	Outel

Figure 11: Example policies for the Rainforest application.

Carol's code would rerun up to line 4 or 6 until it could precommit, and then the remainder of the transaction would run.

8. EVALUATION

To evaluate our implementation, we built three example Fabric applications, and tested them using our modified Fabric compiler:

- an implementation of the hospital example from § 2;
- a primitive blog application (from which Fig. 10 was taken), in which participants write and comment on posts with privacy policies; and
- an implementation of the Rainforest example from § 2.

8.1 Hospital

We implemented the programs described in our hospital example (Fig. 3). In the implementation, Patsy's code additionally appends the addresses of HIV-positive patients to a secure log. In a third program, another trusted participant reads the secure log.

With our changes, the compiler correctly rejects Patsy's code. We amended her code to reflect Fig. 4. Of the 350 lines of code, we had to change a total of 113 to satisfy relaxed monotonicity and compile. Of these 113 lines, 23 were additional method annotations and the remaining 90 were the result of refactoring the transaction that retrieves the addresses of HIV-positive patients. SC scheduled the transactions without leaking information. The patient's HIV status made Mallory neither more nor less likely to receive aborts.

8.2 Blog

In our primitive blog application, a store holds API objects, each of which features blog posts (represented as strings) with some security label, and comments with another security label. These labels control who can view, edit, or add to the posts and comments.

In one of our programs, the blog owner atomically reads a post and updates its text to alternate between "fizz" and "buzz". In another program, another user comments on the first post (Fig. 10). To keep this comment pertinent to the content of the post, reading the post and adding the comment are done atomically. Since posts and comments have different labels, this transaction has at least two stages: one to read the post, and another to write the comment.

We were able to compile and run these programs with our modified system with relatively few changes. Of the 352 lines of code, we had to change a total of 50, primarily by adding annotations to method signatures (§ 7.2.4).

8.3 Rainforest

We implemented the Rainforest example from § 2.1. In our code, two nodes within Rainforest act with Rainforest's authority. They perform transactions representing the orders of Gloria and Fred from Fig. 1. Each transaction updates inventory data stored at one location, and banking data stored at another. Fig. 11 gives examples of the policies for price, inventory, and banking data.

While attempting to modify this code to work with SC, we discovered that the staging order chosen in § 7.2.3 makes it impossible to provide the atomicity of the original application while both meeting its security requirements and ensuring deadlock freedom.

To illustrate, suppose Gloria is purchasing an item from Outel. To ensure she is charged the correct price, the event that updates

| Example | Program | SC | | | 2PC |
		# stages	Dyn. checks	Total time	Total time
Hospital	patsy	3	0.45 ms	9.17 ms	6.38 ms
Blog	post	2	0.11 ms	1.03 ms	1.01 ms
	comment	3	0.29 ms	1.30 ms	1.01 ms

Figure 12: Performance overhead of SC. Reported times are per-transaction averages, across three 5-minute runs of the blog application and three 20-minute runs of the hospital application. Relative standard error of all measurements is less than 2%.

the inventory must share a transaction with the one that debits Gloria's bank account. The conflict label for the inventory event corresponds to {Outel}, whereas the conflict label for the debit event corresponds to {Bank, Gloria}. Since neither is a subset of the other, the compiler cannot put them in the same transaction.

These difficulties in porting the Rainforest application arise because Fabric is designed to be an open system, and so an *a priori* choice of staging order must be chosen. If the application were written as part of a closed system, deadlock freedom can be achieved by picking a staging order that works for this particular application (e.g., {Outel} before {Bank, Gloria}), but it might be difficult to extend the system with future applications.

8.4 Overhead

The staged commit protocol adds two main sources of overhead compared to traditional 2PC. First, each stage involves a round trip to prepare the data manipulated during the stage, leading to overhead that scales with the number of stages and with network latency. Second, as described in § 7.3, dynamic labels result in potential stagepoints, which must be resolved using run-time checks. The number of checks performed depends on how well the compiler's static analysis predicts potential stagepoints.

We measured this overhead in our implementation on an Intel Core i7-2600 machine with 16 GiB of memory, using the transactions in our examples. The post and comment transactions in the blog example were each run continually for 15 minutes, and Patsy's transaction in the hospital example was run continually for 1 hour.

Fig. 12 gives the overall execution times for both the original system and the modified system. For the modified system, it also shows the number of stages for each transaction and the average time spent in dynamic checks for resolving potential stagepoints. The comment transaction in our experiments has one more stage than as described in Fig. 10, because in all transactions, there is an initial stage performed to obtain the principals involved in the application.

By running the nodes on a single machine and using in-memory data storage, we maximize the fraction of the transaction run time occupied by dynamic checks. Nevertheless, this fraction remains small. While the effective low latency of communication between nodes reduces the overhead due to communication round-trips for staging precommits, we report the number of stages, from which this overhead can be calculated for arbitrary latency.

9. RELATED WORK

Various goals for atomic transactions, such as serializability [28] and ACID [20], have long been proposed and widely studied, and are still an active research topic. While much of the recent interest has been focused on performance, we focus on security.

Information leaks in commonly used transaction scheduling protocols have been known for at least two decades [36, 6]. Kang and Keefe [21] explore transaction processing in databases with multiple security levels. Their work focuses on a simpler setting with a global, trusted transaction manager. They assume each transaction has a single security level, and can only "read down" and "write up." Smith et al. [36] show that strong atomicity, isola-

tion, and consistency guarantees are not possible for all transactions in a generalized multilevel secure database. They propose weaker guarantees and give three different protocols that meet various weaker guarantees. Their Low-Ready-Wait 2PL protocol is similar to SC, and provides only what the authors call ACIS$^-$–correctness. Specifically, "aborted operations at a higher level may prevent all lower level operations from beginning" [36, p37]. Although our implementation is conservative and would not allow such a thing, the theory behind SC could allow a later stage with less trustworthy participants to hold up earlier, precommitted stages indefinitely. Duggan and Wu [16] observe that aborts in high-security subtransactions can leak information to low-security parent transactions. Their model of a single, centralized multilevel secure database with strictly ordered security levels is more restrictive than our distributed model and security lattice. Our abort channels generalize their observation. They arrive at a different solution, building a theory of secure nested transactions. Atluri, Jajodia, and George [5] describe a number of known protocols requiring weaker guarantees or a single trusted coordinator. Our work instead focuses on securely serializing transactions in a fully decentralized setting. Our analysis is also the first in this vein to consider liveness: SC can guarantee deadlock freedom of transactions with relaxed monotonicity.

In this work, we build on a body of research that uses lattice-based information flow labels and language-based information flow methods [12, 14, 31]. Relatively little work has studied information flow in transactional systems. Our implementation is built on Fabric [24, 3], a distributed programming system that controls information flow over persistent objects. The only other information-flow-sensitive database implementation appears to be IFDB [33], which also does not account for abort channels.

10. CONCLUSION

There is a fundamental trade-off between strong consistency guarantees and strong security properties in decentralized systems. We investigate the secure scheduling of transactions, a ubiquitous building block of modern large-scale applications. Abort channels offer a stark example of an unexplored security flaw: existing transaction scheduling mechanisms can leak confidential information, or allow unauthorized influences of trusted data. While some sets of transactions are impossible to serialize securely, we demonstrate the viability of secure scheduling.

We present relaxed monotonicity, a simple condition under which secure scheduling is always possible. Our staged commit protocol can securely schedule any set of transactions with relaxed monotonicity, even in an open system. To demonstrate the practical applicability of this protocol, we adapted the Fabric compiler to check transactional programs for conditions that allow secure scheduling. These checks are effective: the compiler identifies an intrinsic security flaw in one program, and accepts other, secure transactions with minimal adaptations.

This work sheds light on the fundamentals of secure transactions. However, there is more work to be done to understand the pragmatic implications. We have identified separate necessary and sufficient conditions for secure scheduling, but there remains space

between them to explore. Ultimately, abort channels are just one instance of the general problem of information leakage in distributed systems. Similar channels may exist in other distributed settings, and we expect it to be fruitful to explore other protocols through the lens of information flow analysis.

Acknowledgments

11. REFERENCES

[1] Distributed transactions: .NET framework 4.6. https://msdn.microsoft.com/en-us/library/ms254973%28v=vs.110%29.aspx. Accessed: 2015-11-13.

[2] XA standard. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 3571–3571. Springer US, 2009.

[3] O. Arden, J. Liu, T. Magrino, and A. C. Myers. Fabric 0.3. Software release, http://www.cs.cornell.edu/projects/fabric, June 2016.

[4] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, 2010.

[5] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Advances in Database Systems. Springer US, 2000.

[6] V. Atluri, S. Jajodia, T. F. Keefe, C. D. McCollum, and R. Mukkamala. Multilevel secure transaction processing: Status and prospects. *DBSec*, 8(1):79–98, 1996.

[7] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, May 2006.

[8] E. Bertino, B. Catania, and E. Ferrari. A nested transaction model for multilevel secure database management systems. *ACM Trans. Inf. Syst. Secur.*, 4(4):321–370, Nov. 2001.

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage. In *SOSP*, 2011.

[10] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *CSF*, 2008.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[12] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[13] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[14] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[15] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP*, 2015.

[16] D. Duggan and Y. Wu. Transactional correctness for secure nested transactions. In *TGC*, pages 179–196, 2011.

[17] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. of the ACM*, 19(11):624–633, Nov. 1976. Also published as IBM RJ1487, December 1974.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985. (MIT/LCS/TR-282).

[19] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[20] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[21] I. E. Kang and T. F. Keefe. Transaction management for multilevel secure replicated databases. *J. Comput. Secur.*, 3(2-3):115–145, Mar. 1995.

[22] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *2009 IEEE Computer Security Foundations*, July 2009.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[24] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP*, 2009.

[25] D. McCullough. Noninterference and the composability of security properties. In *IEEE Symp. on Security and Privacy*, pages 177–186. IEEE Press, May 1988.

[26] S. Microsystems. JavaBeans (version 1.0.1-a). http://java.sun.com/products/javabeans/docs/spec.html, Aug. 1997.

[27] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000.

[28] C. H. Papadimitriou. The serializability of concurrent database updates. *J. of the ACM*, 26(4):631–653, Oct. 1979.

[29] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012.

[30] A. W. Roscoe. CSP and determinism in security modelling. In *IEEE Symp. on Security and Privacy*, 1995.

[31] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[32] A. Sabelfeld and A. C. Myers. A model for delimited release. In *2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191. Springer-Verlag, 2004.

[33] D. A. Schultz and B. Liskov. IFDB: decentralized information flow control for databases. In *EUROSYS*, 2013.

[34] I. Sheff, T. Magrino, J. Liu, A. C. Myers, and R. van Renesse. Safe serializable secure scheduling: Transactions and the trade-off between security and consistency. Technical Report 1813–44581, Cornell University Computing and Information Science, Aug. 2016.

[35] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. Windows XP update. Wiley, 2003.

[36] K. Smith, B. Blaustein, S. Jajodia, and L. Notargiacomo. Correctness criteria for multilevel secure transactions. *Knowledge and Data Engineering, IEEE Transactions on*, 8(1):32–45, Feb 1996.

[37] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW*, 2003.