TinyTX

Isaac Sheff

February 13, 2016

Abstract

The distributed systems upon which modern life depends, from medical record storage to Facebook, are built upon shaky foundations. No programming language, framework, or system designed for distributed applications posses both the flexibility necessary to express complex protocols, and strong security guarantees. I propose TinyTX: a new programming language and runtime which possesses both the strong distributed system guarantee of serializability, and the strong security guarantee of non-interference. TinyTX is built around a new primitive, the Single Write-Site Transaction (SWST), which I believe to be a natural and powerful building-block for complex protocols. This language is not specifically designed to be intuitive for programmers. Instead, we will provide a solid foundation to which the distributed programs of the future will compile, enabling strong analysis of security, correctness, and optimization.

Contents

1 Motivation

Many of today's most popular applications, such as Google, Facebook, or Twitter, are fundamentally distributed: they cannot be run entirely on one machine. Not only would it be far too much work for one computer, but these applications involve communication from one person's machines to another. Every year, new reports arise of popular applications failing or leaking private data because of some subtle design flaw or security oversight. Programming languages can provide the tools to solve these problems: not just to patch new holes as they are discovered, but to make programs secure and correct *by design*.

2 Problem

We need programming languages that make it easy to reason about the security and correctness of high-performance distributed systems. While a number of languages and frameworks exist to build specific kinds of distributed systems, or perform certain kinds of operations [60, 25], each has its own limitations, and demands a complex analysis. Often these languages are too restrictive in their abilities or too focused on some application to model complex distributed protocols such as Consensus [12, 52] efficiently. No programming language presents a sound framework for constructing low-level distributed protocols that are easy to reason about both in terms of security and correctness.

Looking to address these problems, we find there are many questions left unanswered:

- What is the natural building block for distributed protocols?
- How can that block be implemented?
- What are its security implications?

At first blush, it might seem that the natural building block, the minimal unit from which we build everything else, would be the *message*: a piece of information transferred from one site (computer, server, machine, location) to another. Yet storage is another critical component of distributed systems: they must remember information. What's more, they must be able

to compute functions on information stored: otherwise, such a system can *only* store and replicate data. Messages do not capture this.

Perhaps, then we need a different building block, one which reads information from storage at one site, sends it to at another, computes some function on the information, and then store the results. How should we set up this information storage? What exactly can a block read? The simplest answer would be to assume each site stores some set of data, each uniquely identified with a key, and a block reads from one key and writes to another. However, distributed systems must be able to compute functions that take into account multiple data: we might want to calculate, say, the average grade in a class using a system that stores each student's grade.

Therefore we propose the Single Write-Site Transaction (SWST) as a natural basis for distributed protocols. Intuitively, each SWST reads data (potentially located on disparate sites), computes some functions on this data, and writes the output to some destination site. The details we leave to section 3. With these as our core unit, we introduce TinyTX.

SWSTs have a fairly straightforward security requirement: all the data used as the input of a function must be permitted to influence the Data written as a result of that function. In line with prior work on Information Flow based security [26, 36, 42, 93, 67], we assign each datum a security label, which describes which other data it is permitted to affect. Our language thus tracks the flow of information, and programs allowing insecure flows are considered incorrect.

To make reasoning about SWSTs as straightforward as possible, we'll make them fully **serializable**. Serializability is a popular abstraction that can help make distributed programs easier to reason about. Each program is divided into *transactions*, (in our case SWSTs), each of which can be thought of as executing *atomically*: any results observed should look as if one transaction were executed after another, in some "serial" ordering. [69]. Another advantage of SWSTs as a building block is that unlike traditional transactions, SWSTs can in the best (and hopefully most popular) case be committed with only a single message send from wherever Data is read to where it is written [34]. This makes them an extremely fast, efficient base unit from which to construct distributed programs.

3 Approach

TinyTX will be a new, low-level language for distributed systems, featuring information-flow based enforcement of non-interference. Unlike most prior languages, it is not our goal to build a language for the convenience of the programmer, but rather for convenient analysis: TinyTX will express distributed programs and protocols efficiently, and in a way that facilitates reasoning about security and correctness. It will also form a sound basis for constructing and reasoning about higher levels of abstraction.

The most important unit in this language is the Single Write-Site Transaction (SWST). These are terminating transactions which begin with a predetermined set of Data they read, a set they write, and total functions to derive the latter from the former. Furthermore, all the written Data must be at the same site (which can be thought of as a node, location, machine, server, or store). This makes it possible (in the best case) for SWSTs to commit after only one message send from the site of Data read to the site of Data written. These SWSTs will be fully serializable, and I believe will serve as powerful building blocks for larger, more complex protocols.

For example, we can create an *abstract site*, which is itself composed of other sites, and whose basic operations are defined with SWSTs. It might represent a storage location with more availability, or storage space than any constituent site could provide. We can then write SWSTs that write to the abstract site.

Programs written in our language will consist of a set (possibly infinite: there might be, say, one for each integer) of such SWSTs. The compiler will analyse this set to determine which SWSTs, under which circumstances, conflict with which others. Ideally, one SWST would have to wait for another's completion only when absolutely necessary.

This is also a Security-Typed Information Flow language seeking to preserve Non-Interference: Each Datum has a security policy limiting which other Data it may affect. This can be used to model a wide variety of security-related issues, including failure-tolerance [80]. Non-Interference requires that for any observer, and any event (practically speaking, in our case, SWLT), the observer's view of the system after the event is solely determined by its view before the event. No information which is barred from flowing to the observer should influence any outcome it can see.

3.1 Primitives

3.1.1 Data d

I assume the existence of some type Data. Storage sites will store these Data in key:value pairs.

I assume that Data include unique nonces (as in, when one is generated, we can be sure it will not be generated independently elsewhere). These allow Data to serve as unique identifiers for other Data entries. I also assume that Data include lists, sets, and tuples of Data, as well as some manner of $NULs$ value.

$$d := [d] \mid \{d\} \mid \langle d, d, \ldots, d \rangle \mid nonce \mid NULL \mid \ldots$$

3.1.2 Labels ℓ

Associated with each Key, stored Datum, or variable of type Data is a label. If a datum or variable d has label ℓ, we write $d : \ell$. Labels act a lot like types.

Labels express whether one datum is permitted to influence another. When a datum with label ℓ can influence a datum with label ℓ', we write $\ell \sqsubseteq \ell'$. This property is transitive: $\ell \sqsubseteq \ell' \sqsubseteq \ell'' \Rightarrow \ell \sqsubseteq \ell''$. Therefore $\sqsubseteq$ acts a lot like subtyping: if $\ell \sqsubseteq \ell'$, then wherever a Datum with label ℓ' can be used, a Datum of label ℓ can be used instead.

$$\frac{d : \ell \qquad \ell \sqsubseteq \ell'}{d : \ell'}$$

Furthermore, each site s has two associated labels: $outbound(s)$ and $inbound(s)$:

- $outbound(s)$ represents the lower bound on data flowing *from* site s. If some datum with label ℓ could have been influenced by s (such as if it were stored on s) then $outbound(s) \sqsubseteq \ell$.
- $inbound(s)$ represents the upper bound on data which may flow *to* site s. If some datum with label ℓ might influence data on site s, then $\ell \sqsubseteq inbound(s)$.

These labels are prescriptive: they are parameters of the system describing the security clearances and reliability of the sites. Therefore, if some piece of information with label ℓ is stored on a site s, then s can influence the information, so we require $outbound(s) \sqsubseteq \ell$. Additionally, s would have

access to the information, so we require $\ell \sqsubseteq inbound(s)$. Putting these two together, it is clear that for any site which stores any information, we require $outbound(s) \sqsubseteq inbound(s)$, in essence that information may flow from s to itself.

We assume labels form a lattice, where $\ell \sqsubseteq \ell \sqcup \ell'$, $\ell' \sqcap \ell \sqsubseteq \ell$, and both $\sqcup$ and $\sqcap$ are commutative. As shown in prior work, a lattice of labels can express rich availability, integrity, and confidentiality policies.

3.1.3 Keys k

Data stored are identified by a key containing a site (considered to be the "primary site" at which this Datum is stored), a label representing the security of that Datum, and a unique identifier, which is another Datum. Using an arbitrary Datum as an identifier is reminiscent of traditional Hashtables, in which more or less arbitrary objects can be used as keys.

$$k := \langle s,\ \ell,\ d \rangle$$

The label of the key itself is equivalent to that of the identifier Datum it contains:

$$\langle s, \ell, d_k \rangle : \ell \Leftrightarrow d_k : \ell$$

If a Key $\langle s, \ell, d_k \rangle$ is used to store a datum d_v, this effectively requires $d_v : \ell$. The label ℓ is meant to represent the security level of information stored with this key. As a result, multiple different Data may be stored under keys which differ only in the label component. This in some ways resembles Secure Multi-Execution systems, in which different versions of Data are kept for each security level [28, 39]. This also avoids the "aliasing problem" encountered in some prior Information Flow storage systems [**?**]. Because the label of the Datum stored is *part of* the key, it is not possible for two different entities to attempt to store data with different security levels under the same key.

3.1.4 Functions F

We assume the existence of *total* functions $F : d^n \to d$. Reasoning about non-termination is hard, so we'll skip that for now. These functions map some number of Data arguments to a Datum return value. The label of the return value is the $\sqcup$ of all the arguments' labels.

$$F : d : \ell_1 \times d : \ell_2 \times \cdots \times d : \ell_n \ \to \ d : \ell_1 \sqcup \ell_2 \sqcup \ldots \sqcup \ell_n$$

The following (relatively standard) subtyping rule applies to the labels of inputs and outputs:

$$\frac{F : (d : \ell_i)^n \to d : \ell_F \qquad \forall i. \ell'_i \sqsubseteq \ell_i \qquad \ell_F \sqsubseteq \ell'_F}{F : (d : \ell'_i)^n \to d : \ell'_F}$$

We also assume allow total functions with return type *Boolean*, acting as predicates. We can calculate labels for these *Boolean*s as if they were Data.

3.1.5 Timestamps t

Data stored is timestamped. Timestamps represent a logical time which is monotonically increasing. It may be helpful for optimization purposes for this to correlate in some way to clock time.

Timestamps have equality $=$ and comparison $<$ operations
(with $t_1 \leq t_2 := (t_1 = t_2) \vee (t_1 < t_2)$). Timestamps also have an *increase* operation $incr : t \to t$, such that $t < incr(t)$.

3.1.6 Transaction Identifiers *tid*

At the launch of each transaction, a unique *tid* is generated for that specific instance of that transaction. These are storable, totally ordered, and each encodes a site representing where the transaction writes. This site is written $s(tid)$.

*tid*s are totally ordered largely for deadlock-breaking purposes: as in the distributed dining philosophers' problem, it is advantageous when breaking cycles to have ordered identifiers [16].

3.1.7 Sites s

Sites, intuitively, represent physical machines which store data. Each functions as a key:value store.

A site s specifically stores key:value pairs

$$\langle s', \ell_k, d_k \rangle : \langle d_v, t, \{k\}, \{tid\} \rangle$$

Which associate each stored Key with:

- a datum
- a timestamp

- **dependencies**: a set of keys representing other data which have affected this datum in the past

- **read_locks**: a set of transaction identifiers representing which transactions have read this datum, but may not have committed yet. **read_locks** themselves have no labels. They are used for serializing SWSTs, but never communicated to another site. Thus, the only site that learns the **read_locks** is precisely that site which has taken part in all the transactions listed, and we need not worry about **read_locks** leaking information.

Furthermore, it is assumed that all such pairs are cryptographically signed by site s'. In this way, they can be transmitted via roundabout channels, and still maintain integrity.

If a site s stores a pair with a key $\langle s', \ell, d_k \rangle$, then it is required that $outbound(s') \sqsubseteq \ell \sqsubseteq inbound(s)$, since the Datum can flow from site s' and has flowed to s.

If a site s stores a key:value pair $\langle s', \ell, d_k \rangle : \langle d_v, t, \{k\}, \{tid\} \rangle$ such that $s = l'$, then the pair is said to be *stored* at s. s' can be called the *primary store* for that pair. Otherwise, the pair is said to be *cached* at s.

Sites maintain Multiversion Concurrency Control with Read Locks, as it will allow single write-site transactions to maintain serializability with the fewest message-sends in the best case. Intuitively, each time a stored value is changed, all values upon which that change depends are cached at its site, and only the highest-timestamped cached values are used. As part of Multiversion Concurrency Control, sites cache all information which has affected information they store. This is part of preventing inconsistent (non-serializable) updates. While this caching process can seem extreme, there are well-known optimizations which result in not only acceptable, but excellent performance in MCC systems [61].

To prevent one transaction writing "while" some value upon which it is dependent changes, transactions read-lock values until their completion. In the best case, wherein the value to be written is not read-locked, only one round of messages must be sent before transaction commit, with another afterwards to remove read-locks.

A site s must provide a few basic operations:

- **remove_tid**: $k \times t \times tid \to ()$

If a tuple is stored with the given key and timestamp, and if that tuple's **read_lock** *tid* set contains the given *tid*, remove it from the set.

- **get**:$\langle s', \ell_k, d_k \rangle : \ell \times tid \rightarrow \langle d_v, t, \{k\}, \{tid\} \rangle : \ell \sqcup \ell_k$

  Get takes in a key, and returns the most recently written stored value affiliated with that key. To simplify semantics, if there is no such pair, one is created and stored with a $NULL$ datum, some initial timestamp, and empty sets.

  The resulting key:value pair's signature (signed by s') is assumed to be retrievable. It can be stored with the tuple. For tuples with a $NULL$ datum, an initial timestamp, and empty sets, this signature is not necessary. All tuples can be safely said to have been $NULL$ at "initial time."

  Finally, the input *tid* is also inserted into the stored tuple's **read_lock** *tid* set. This represents a transaction reading a value, but not yet committing.

  The label of **get**'s return value is the $\sqcup$ of the label *in* the key, representing restrictions on the datum stored, and the label *of* the key, representing restrictions on data flowing to this access. This prevents the result of a **get** from flowing to anyone not cleared to receive information about the data stored, or the fact that the access happened.

- **put**: $\langle s', \ell_k, d_k \rangle : \ell \times [\langle k : \ell_i, d : \ell'_i, t \rangle] \times (APPEND \mid t) \times F \times \{tid\} \rightarrow Bool$

  Put takes in a key, a list of timestamped dependencies (including key, value, and timestamp each), either a timestamp or the APPEND flag, a function, and a set of *tid*s. The set of *tid*s can be thought of as representing the transaction executing the **put**, and possibly others with which it commutes. It returns a Boolean, representing whether or not it was successful.

  In essence, **put** computes F using some of the dependencies as input, and stores the result under the key given.

  In order for **put** to label-check (be secure), some basic information flow properties must hold. Ideally, these would be checked at compile time.

– The fact that this write is occuring must be permitted to affect the information stored

$$\ell \sqsubseteq \ell_k$$

– The Datum being stored must be permitted to flow to the site executing the **put**:

$$\ell_k \sqsubseteq inbound(s)$$

– The site associated with the datum being stored, s', must be sufficiently trusted to provide data with the given label:

$$outbound(s') \sqsubseteq \ell_k$$

– If F is a function of n inputs, then the first n dependencies in the list must be permitted to flow to those inputs. Furthermore, the output of F must flow to the Datum stored, and with the given label:

$$F : (d : \ell_i \sqcup \ell'_i)^n \rightarrow \ell_k$$

If s already has a stored value for the input key, with a timestamp greater than the input one, it does nothing, and returns $True$.

Otherwise:

For each dependency, Put updates s's stored key:value pair with the new datum iff it either has no prior stored value for that key, or has one with an older timestamp. These new, updated stored values will have empty **read_locks**. This caching process is part of Multiversion Concurrency Control, and ensures that writes to this site are based on a consistent view of the system.

Put then evaluates F, using as input its stored Data for each dependency. If the result is equal to the stored value's Datum, nothing happens, and **put** returns $True$.

If the result isn't equal to the stored value's Datum, and the stored value's **read_locks** have *tid*s not in the *tid* set input to **put**, then the value cannot be changed, so **put** returns $False$. Otherwise, the result is stored in place of the old stored Datum.

Put then increments (or initiates, if necessary) the timestamp if APPEND is input, and otherwise updates it to the input timestamp. Put updates the dependency key set with the union of the old set, the set

of keys in the dependency list, and each of their dependencies in their stored entries on s. Finally, **put** returns $True$.

Optimization: Dependencies are transitive, and as a result, not all of a **put**'s dependencies will be inputs to F. For syntactic purposes, if F has n inputs, the first n dependencies are the ones to be input to F. For some non-input dependencies, it is efficient to store a "$FETCHME$" flag instead of a datum. In the event this datum needs to be read, a **get** must be performed. In the event that such a get retrieves a datum with an earlier timestamp, it must be repeated.

Sites themselves are assumed to be capable of serializing *fail-able* transactions featuring sets of these operations locally. Specifically, local transactions should be able to include **get**ting some values, and then using Data from those in a list of subsequent **put**s. However, in the event that a **put** in a transaction returns $False$, the transaction aborts, and no effects of the transaction persist. This is the case in which the transaction *fails*. Otherwise, it succeeds.

3.1.8 Single Write-site Transactions Δ

The conceptual building-block of programs in TinyTX is the Single Write-site Transaction. Each Single Write-site Transaction has 4 components:

- A set of keys representing data read in the transaction. Each key (index i) naturally contains a label ℓ_i representing the security of the datum. Each key is also itself labelled. The label ℓ'_i of the key itself represents the security of the *fact that the read occurs*. Influencing the key influences what is read and where, and learning of it represents learning of the existence of the read. Therefore, we require that ℓ'_i flows to the site at which the read occurs: $\ell'_i \sqsubseteq inbound(s_i)$.

$$inputs_\Delta := \{\langle s_i, \ell_i, d_i\rangle : \ell'_i\}$$

- A site at which the write occurs, called s_w.

- A set of values (index k) to write. Each value to write is expressed as a function, with input being some subset of the data read. Paired with each function is a key, with site s_k, data label ℓ_k. The key itself has a label ℓ'_k.

$$outputs_\Delta := \{\langle F_k : d^n \to d, \quad \langle s_k, \ell_k, d_k\rangle : \ell'_k\rangle\}$$

Like with the reads, the fact that a write occurs has the security level of its key. For serializability, all writes must happen after all reads, and so the fact that a write occurs implies that all reads occurred:

$$\ell'_i \sqsubseteq \ell'_k$$

Naturally, the fact that a write occurs influences the value written:

$$\ell'_k \sqsubseteq \ell_k$$

- A set of "children:" Single Write-site Transactions which launch *after completion* of this transaction (index c). Conceptually, each of these transactions will serialize after this transaction. They function as continuations for the purposes of constructing larger programs. There may be other, arbitrary transactions serialized in between.

  Paired with each child is a predicate on the data read which determines whether or not to launch the child transaction.

$$children_\Delta := \{\langle F_c : d^n \to Bool : \ell_c, \quad \Delta_c : \ell_c \rangle\}$$

  The labels on the keys used by the child transaction, representing the security of "whether or not the events occur," impose a security restriction on its predicate. All the labels of the reads which input to the predicate must flow to the labels of the keys in the child transaction. We write the $\sqcap$ of all the labels of the keys in a transaction Δ as ℓ_Δ. Therefore:

$$\ell'_k \sqsubseteq \ell_c$$
$$\ell'_i \sqsubseteq \ell_c$$

  The site of the writes is the only one that can be sure of knowing all inputs. It must therefore be responsible for launching child transactions. Security requires:

$$outbound(s_w) \sqsubseteq \ell_c$$

  As the values input to the predicate influence whether or not the child happens, the output label of the predicate must flow to ℓ_c.

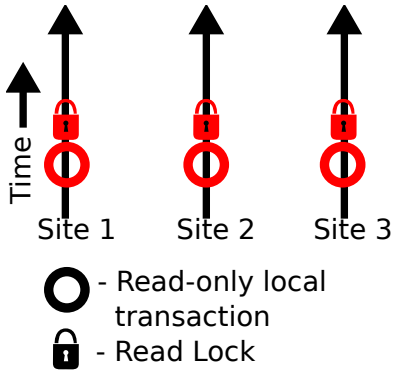
It may seem that the value of all reads from the transaction flow to children, since children are serialized after all reads, and writes dependent upon those reads. This is not necessarily the case. With the "$FETCHME$" optimization from **put**'s definition, the value of all writes need not be determined prior to launching these children, only timestamps and keys.

$$\Delta := \left\langle \begin{array}{l} inputs_\Delta, \\ s_w, \\ outputs_\Delta, \\ children_\Delta \end{array} \right\rangle : \ell'_k \bigsqcap \ell'_i$$

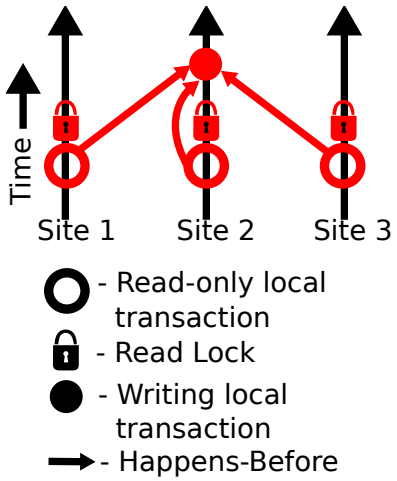
3.2 Under the Hood

In implementation, TinyTX would run each single write-site transaction as one transaction per site involved:

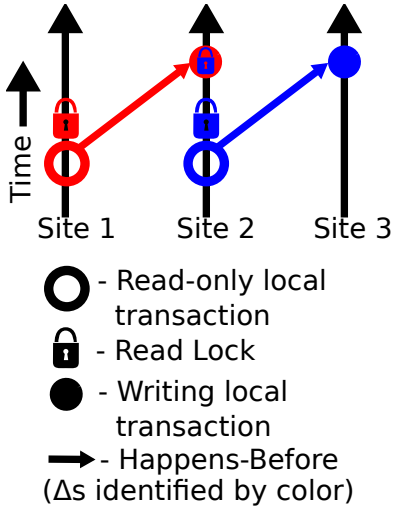
- First, all the non-write sites run a local transaction consisting of one get for each read key at that site. This would place read-locks with this transaction's *tid* on all values read in this transaction.



- Second, the write site runs a local transaction including one get for each read key at that site, and one put for each write.



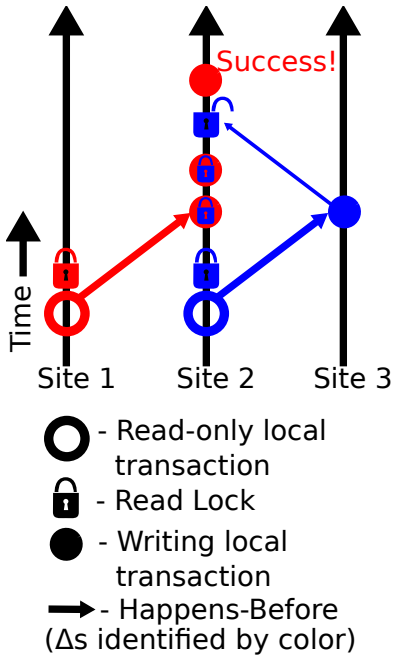
If the local transaction fails, then one or more **put** is writing a value which is presently read-locked by another SWST.



One traditional solution would be to issue **remove_tid**s (via messages to other sites, if necessary) for all **get**s performed in the entire SWST, and restart the entire SWST. This runs the risk of the "abort channel[1]."

Instead, we retry the final local transaction until it succeeds. If there is no deadlock, then eventually read locks will be lifted, and the transaction will succeed.

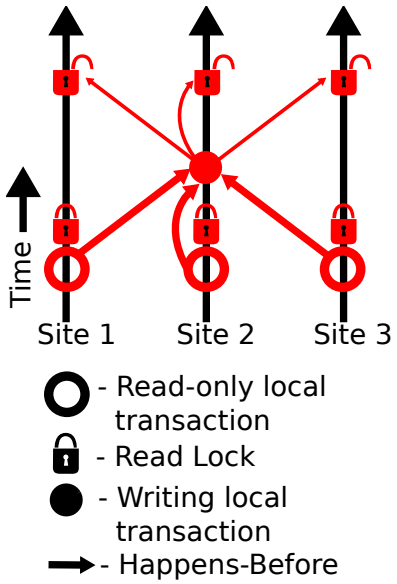
[1]Abort Channels are a major subject of prior work (yet to be published) with Tom Magrino, Robbert van Renesse, Andrew Myers, and myself



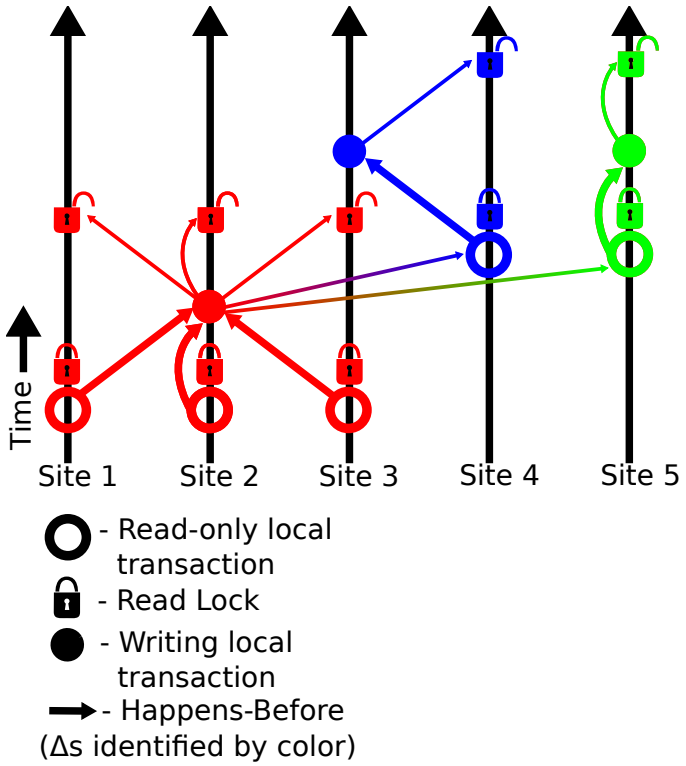
For now, I'll avoid addressing livelock, wherein frequent reads ensure that a datum is continuously read-locked by an ever-changing set of SWSTs. In principle, this can be mitigated by delaying **get** operations.

Deadlock breaking is covered in Section 3.2.1.

- **remove_tid**s are issued (via messages to other sites, if necessary) for all **get**s performed in the entire SWST. This is performed in parallel with the next step.



- Each child SWST is launched iff its predicate evaluates to true. Each value written in this SWST (and its dependencies) is added to the set of dependencies in all writes in all children. This ensures that children are serialized after their parents.



With the "*FETCHME*" optimization for **put**, it may be possible to launch child transactions before all reads and writes have physically completed.

The actual site of computation is left abstract. This is so that it can be chosen by a clever compiler. In principal, however, an SWST could run in minimal rounds as follows: Initiate with some signature proving the integrity of the keys involved, and transmit to all sites involved. The write site could then echo it to the others upon receipt, to ensure all sites involved are aware of the SWST. Each non-writer site executes its local transaction, and transmits the complete results to the write site. The write site begins its local transaction. If the write site's local transaction fails, **get**s are used to determine the blocking read locks, and the local transaction is retried while deadlock-breaking procedures are run (see below). When the local transaction succeeds, the write site calculates the predicates for child SWSTs as inputs become available, launching child SWSTs as necessary. Finally, the write site sends **remove_tid** messages to the read sites, in order to remove read locks for this SWST.

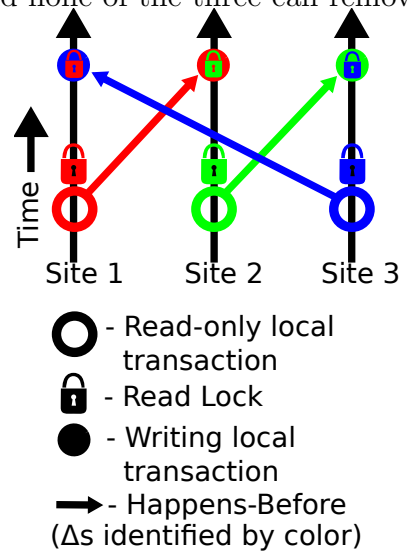
3.2.1 Pop-Up Deadlock Breaking

To avoid actual deadlock without leaking information, I propose a method I call **pop-up** deadlock breaking. I am not aware of an exact pre-existing corollary.

Here I introduce the notion of a **pop-up** message, which is of the form "If read-locks from your SWST were lifted, I might make progress, and then the new values you'd read would be..."

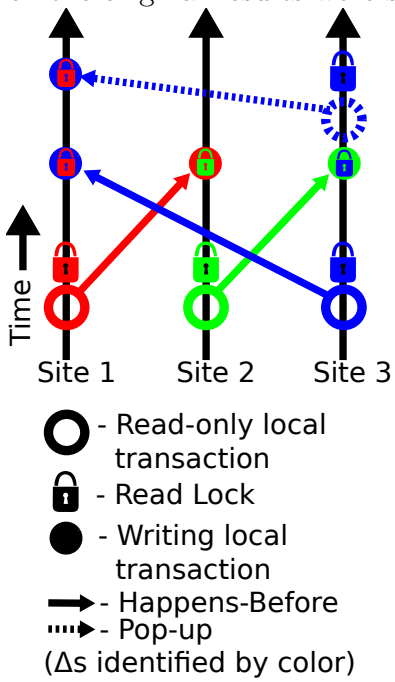
In principal, any deadlock is caused by a cycle of SWSTs, in which each is required to serialize *after* the next. Since it's a cycle, no SWST can go first, and no progress is made.

For example, suppose three SWSTs, Δ_{red}, Δ_{green}, and Δ_{blue} are running at effectively the same time on a system of three sites: called 1, 2, and 3. Δ_{red} reads from site 1, and writes to site 2. Δ_{green} reads from site 1 before Δ_{red} writes there, and so Δ_{green} must be serialized before Δ_{red}. SWST Δ_{blue} reads from site 3, before Δ_{green} writes there, and so Δ_{blue} must be serialized before Δ_{green}. SWST Δ_{blue} also writes to site 1 after Δ_{red} reads there, and so Δ_{red} must be serialized before Δ_{blue}. Thus Δ_{blue} is after Δ_{red}, which is after Δ_{green}, which is after Δ_{blue}, and so none of the three can be serialized first, and none of the three can remove its read locks: deadlock.

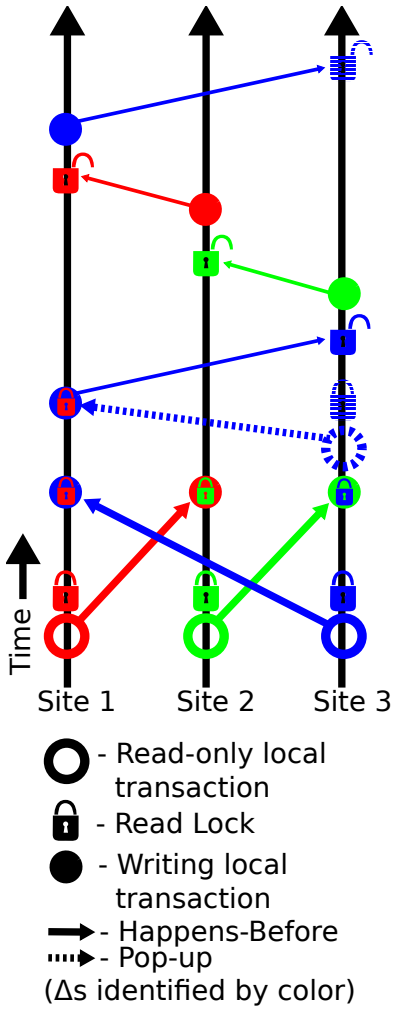


In the event that an SWST's final local transaction fails, it can identify the read locks which are holding it up. In the simplest case, exactly one SWST in each cycle would issue a **pop-up**, in effect delaying itself until after the others. To issue a **pop-up**, the site of a read-only local transaction

calculates what the result of that local transaction *would be* had it occurred after the waiting SWST's local-write began. This creates a read-lock on the hypothetical "future" value. It then transmits the results to the same site to which the original results were sent.



The recipient, upon learning of the "more recent" read value, can lift the old read lock, effectively moving the SWST later in the timeline: it "pops up." With the read-lock lifted, deadlock can be broken, and eventually all SWSTs complete and lift their read-locks, including the one created with the pop-up message.



It is difficult to ensure that exactly one SWST in any cycle will issue a **pop-up**. Most importantly, no two sites "in a row" can effectively issue **pop-up**s, since this can create a situation in which the value transmitted in one **pop-up** is invalidated by another. We can prevent this by requiring that any site that transmits a **pop-up** ignore incoming ones. However, this presents the possibility that everyone in a cycle transmits at once, and so everyone's **pop-up**s are ignored. This we can break with *tid*s. If **pop-up**s are only transmitted from SWSTs with greater *tid*s to SWSTs with lesser ones, some SWST in each deadlock cycle will be least, and can receive a **pop-up** but not send one.

Thus our protocol is as follows: If this SWST's write site has not yet

received a **pop-up** message corresponding to its *tid*, and one such read lock is held by an SWST with a lesser *tid*, then:

- a **pop-up** message is sent to the least read-locking *tid*'s write site explaining what would happen if that read-lock were lifted.
- any incoming **pop-up** messages to this SWST are ignored (queued) until all read-locks with lesser *tid*'s than this SWST's are lifted.

In the event that the read site of a *tid* to whom a **pop-up** was sent lifts its read-lock, a new **pop-up** may be sent. If an SWST which has sent **pop-up**s commits, read locks for the SWSTs to whom it has sent **pop-up**s must be retained for the committed values. In a sense, the reading SWSTs have already read the new, committed values.

Upon receipt of a **pop-up**, an SWST which has not yet committed uses the new values in place of its old values read, and releases the old read-lock. It has by definition acquired a new read-lock on the new values read, which it must release when it commits (or, if it receives a pop-up after committing, when it receives that pop-up). Thus the read action of the SWST "pops up" above (after) the write action of another SWST, which had hitherto been scheduled later.

It is clear that in any deadlock cycle, at least one **pop-up** can be sent. Furthermore, **pop-up**s cannot proliferate around a cycle, and can break deadlock (making one SWST fully after all others in the cycle). No SWST which has sent out a **pop-up** proclaiming what its results will be can receive a **pop-up**, thus changing its behavior and invalidating those results. Thus **pop-up**s preserve correctness and break deadlocks.

Furthermore, **pop-up**s carry no information beyond the values an SWST is already cleared to know. They allow no transmission of information beyond the lifting of read locks (which must happen eventually anyway, and is thus timing information) to flow in any direction contrary to labels. Therefore **pop-up**s are secure.

3.3 Layers of Abstraction

SWSTs can be used to model more complex distributed protocols, including more traditional transactions (which can write to multiple sites), in a way that is easy to reason about, both in terms of security and consistency. For example, we can model more traditional transactions by building *abstract*

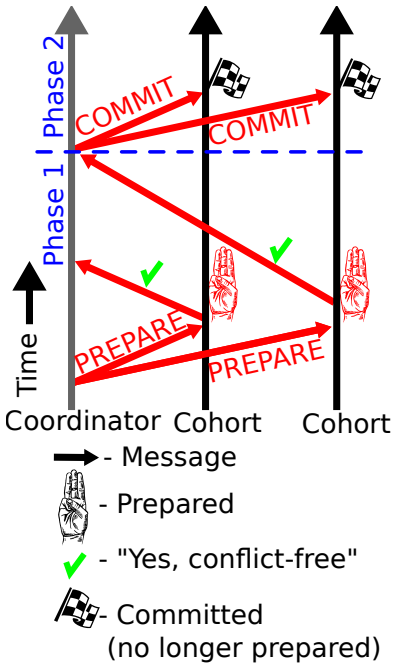
sites, which may be composed of multiple real sites. We have defined the interface which a site must provide, as well as necessary properties of that interface (serializable local transactions featuring sets of puts and gets), and we can simulate that interface with protocols buit from SWLTs for each operation. This allows us to build up larger transactional protocols, represented by SWSTs with abstract write-sites.

Traditional transactions (with multiple write sites) have various procedures for ensuring serializability, even across multiple sites. For example, suppose we have sites *Gloria*, *Harry*, and *Jasmine*. If we want atomic transactions which simultaneously write values to multiple of these, we can create an abstract site incorporating all three, and implement an atomic transactional protocol, such as 2 Phase Commit [34].

In traditional 2 Phase Commit transactions, some site is designated the *coordinator*, and any sites to which information is read or written are called *cohorts*.

- In the first phase, the coordinator sends the cohorts each a "PREPARE" message. Each cohort then determines if it's already *prepared* in a *conflicting* transaction (usually, if one transaction writes a value that the other uses at all, they're "conflicting"). If there are no conflicts, a cohort becomes *prepared* for this transaction, and it sends a "yes" to the coordinator. Otherwise, it sends a "no" to the coordinator.

- In the second phase, if the coordinator receives any "no" responses, it tells all cohorts to abort, and the transaction fails (it may be retried later). If the coordinator receives a "yes" from all cohorts, it sends a "COMMIT" message to all cohorts, and the cohorts can perform any operations in the transaction, and commit them (they become part of the system state). At this point, the cohorts cease to be prepared, as the transaction is complete.



Let's sketch out what that would look like. It would be possible to, say, use *Gloria* as the coordinator. Since we're planning on storing things on all three sites, $Gloria, Harry$, and $Jasmine$ are all cohorts. In this case, each message send in traditional 2PC becomes an SWST.

We shall call our abstract site α. It has a trustworthiness of "all" or "any" of its participants, as:

$$outbound(\alpha) = outbound(Gloria) \sqcap outbound(Harry) \sqcap outbound(Jasmine)$$

$$inbound(\alpha) = inbound(Gloria) \sqcup inbound(Harry) \sqcup inbound(Jasmine)$$

We'll assume that these three participants have some shared secret $SECRET$ which they only use in matter related to α.

Keys of the form $\langle \alpha, \ell, \langle NAME, d \rangle \rangle$ will represent data stored on α which is in fact stored at site $NAME$ (which can be $Gloria$, $Harry$, or $Jasmine$). For completeness, all keys not of that form (which appear in **put**s) are stored on underlying site $Gloria$.

We'll also use tokens like $READ\_LOCKS$ in the keys under we store some of the other things sites must keep track of. For example, $\langle NAME, \ell, \langle SECRET, d, READ\_LOCK$ would be the key under which the **read_locks** which α keeps for key $\langle \alpha, \ell, \langle NAME, d \rangle \rangle$.

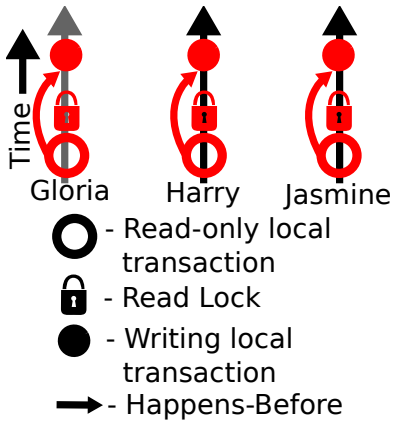
To translate a local transaction of **remove_tid**s, **get**s, and **put**s into an abstract version, we need a notion of "inputs" and "outputs" of basic

operations mapping to "keys" from which SWSTs read and write. Suppose for each instance of a translated basic operation we create new, unique keys at some site (bounded by security constraints) which represent that input and/or output. This is analogous to instantiating variables for function inputs and outputs.

First we have to PREPARE all keys involved. For each site involved, we run an SWLT which checks locally if the relevant keys are already prepared under another transaction, and records whether or not this simulated transaction is prepared. The key for the "whether or not I'm prepared" value should be something like

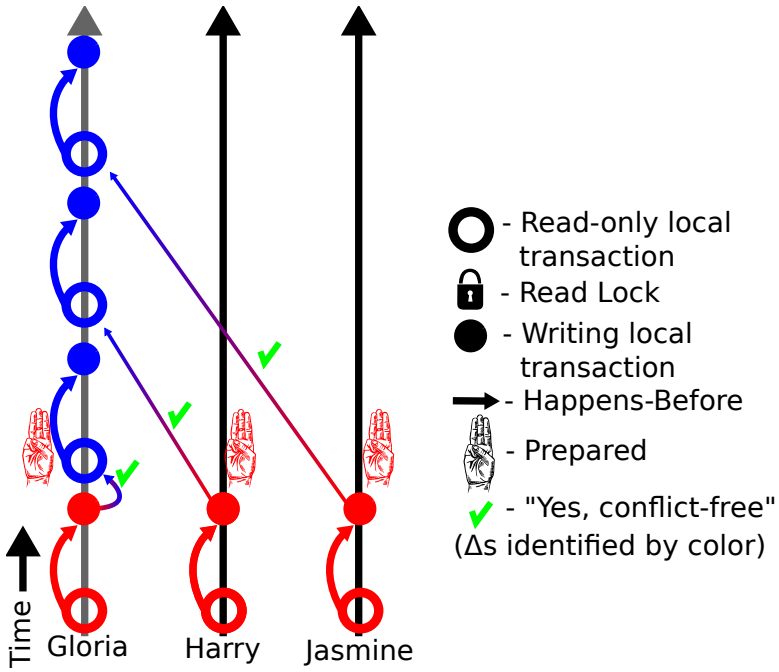
$$\langle\langle NAME, \ell, \langle SECRET, d, PREPARE\rangle\rangle\rangle$$

For each identifier d used in the keys in the simulated transaction.



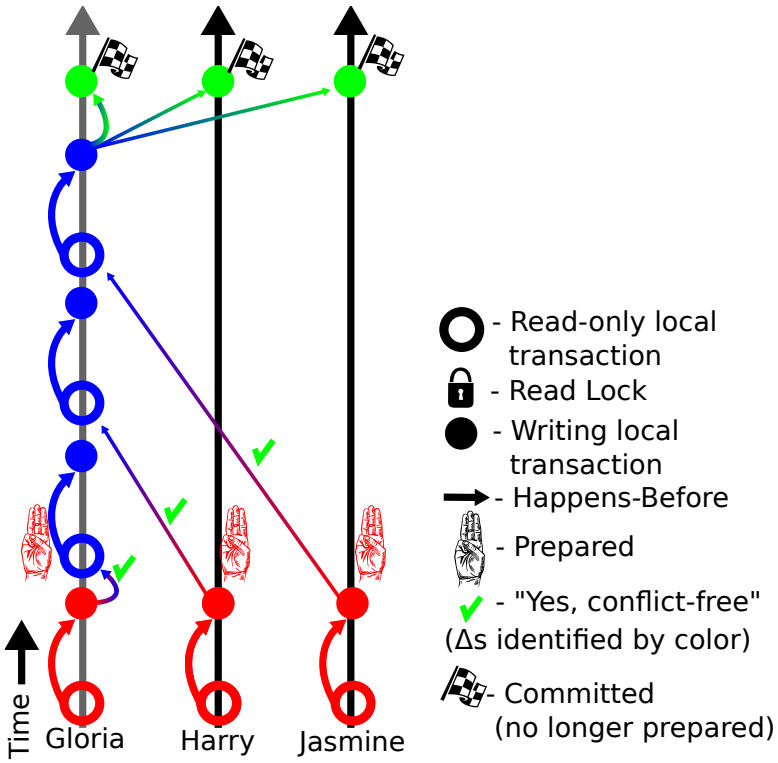
It spawns a child SWST which writes to the coordinator (*Gloria*), recording whether this site can prepare. This completes phase 1. The key for this record might be

$$\langle\langle Gloria, \ell, \langle SECRET, simulated\_transaction\_identifier, NAME\rangle\rangle\rangle$$

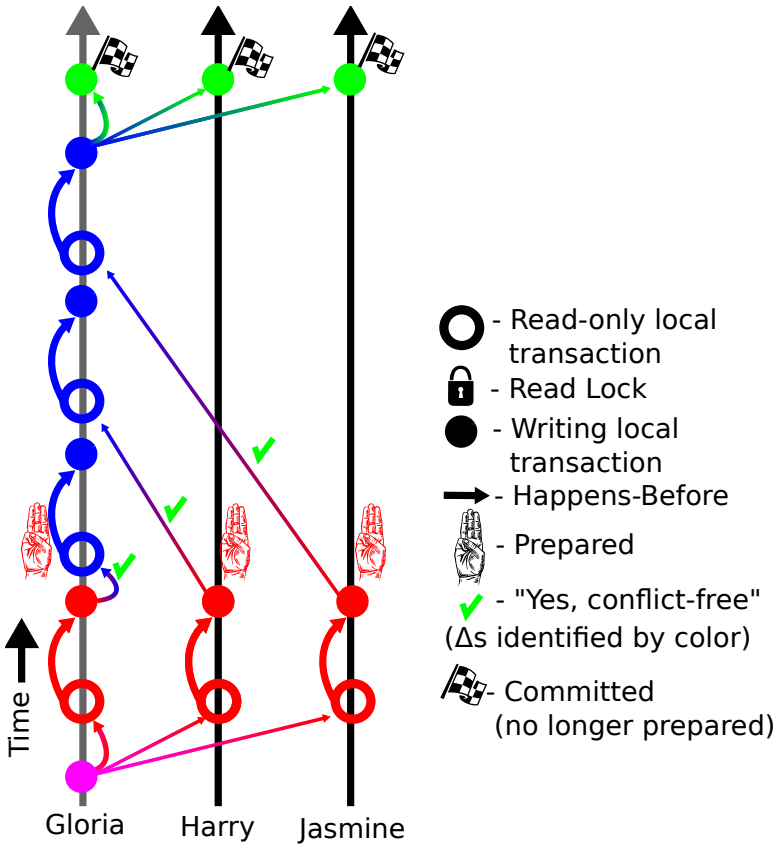


(Here I do not show read locks, but they can be considered to be set just after an SWLT's read and removed just after its write)

Because we know the basic SWSTs from which we build our transaction are strongly serializable, one of these children must go last. Each child can therefore also check the entries left by the other cohorts, and one of them will find that either everyone is prepared, or someone cannot prepare. It then spawns child transactions that either commit (write changes) on all the cohorts, or abort (don't write changes), and removes the $PREPARE$ record.



If we assume some prior SWLT acts as the "start" point, and spawns this whole process, we can see that the actual message-passing pattern is exactly the same as in traditional 2PC.



To properly simulate an abstract site, we must also simulate **remove_tid**$(\langle \alpha, \ell, \langle NAME, d \rangle \rangle, tid)$. This translates simply to an SWST which removes the given tid from the appropriate set of read locks.

$$\left\langle \begin{array}{l} [\langle NAME, \ell, \langle SECRET, d, READ\_LOCKS \rangle \rangle, \quad key(tid)] \\ NAME, \\ \{\langle \lambda xy.x - \{y\}, \quad \langle NAME, \ell, \langle SECRET, d, READ\_LOCKS \rangle \rangle \rangle\}, \\ \ldots \end{array} \right\rangle$$

The child transactions are not defined here, but will be used to "string together" SWSTs into larger operations, much like continuations.

For the **get**s and **put**s of a transaction, things get slightly more complicated.

3.3.1 Analysis

A simple type analysis of this rendering of 2-phase commit demonstrates that the $PREPARE$ flags require information to flow between all pairs of involved sites. What's more, since $PREPARE$ flags influence data written, all sites influence all data written. This brings to light the fundamental hidden channels in 2PC which our prior work on abort channels has begun to uncover.

3.4 Language Extensions

Lantian Zheng introduced the powerful notion of a **message synthesizer** [99, 97]. The idea is that multiple messages can together have security properties no single one can have. A **message synthesizer** waits to receive a set of messages, and then produces a message with a "stronger" label (in some sense).

For example, if a synthesizer awaits three messages, and produces a message when it receives any one of them, then the synthesized message is more *available* than any of the three alone: no one, or even two site failures can stop the synthesized message. On the other hand, if a synthesizer awaits three messages, and only produces a message when it receives all three, *and they're the same*, then the resulting message has more *integrity* than any of the original messages alone. Only an entity who can influence all three of the original messages can influence the synthesized message.

For synthesizers to work, labels must be divided into distinct Confidentiality, Integrity, and Availability components. Specifically, if functions C, I, and A of type $\ell \to \ell$ extract the confidentiality, Availability, and Integrity components, such that

$$\ell = C(\ell) \sqcap I(\ell) \sqcap A(\ell)$$

and

$$\ell_1 \sqcup \ell_2 = (C(\ell_1) \sqcup C(\ell_2)) \quad \sqcap \quad (I(\ell_1) \sqcup I(\ell_2)) \quad \sqcap \quad (A(\ell_1) \sqcup A(\ell_2))$$

Then we can change our language of functions. We can introduce two operations:

- $F_1 \wedge F_2$: If F_1 has n inputs and F_2 has m inputs, $F_1 \wedge F_2$ has $n + m$ inputs. It evaluates F_1 on the first n inputs, and F_2 on the last m inputs. If they return equal values, that value is returned. Otherwise,

$NULL$ is returned. $NULL$ can be thought of as "that datum does not exist at this time."

If F_1 returns a value with label ℓ_1, and F_2 returns a value with label ℓ_2, $F_1 \wedge F_2$ returns a value with label:

$$(C(\ell_1) \sqcup C(\ell_2)) \quad (I(\ell_1) \sqcap I(\ell_2)) \quad (A(\ell_1) \sqcup A(\ell_2))$$

The resulting value has stronger integrity than either F_1 or F_2's outputs alone, because the value is derived from both.

One might think that the confidentialities ought to be likewise $\sqcap$ instead of $\sqcup$. However, reading one, more public value and also reading the $\wedge$ of that public value and a private value would effectively test whether the private value is equal to the public value.

- $F_1 \vee F_2$: If F_1 has n inputs and F_2 has m inputs, $F_1 \wedge F_2$ has $n + m$ inputs. Conceptually, it evaluates F_1 on the first n inputs, and F_2 on the last m inputs. One of the results is then (non-deterministically) returned.

  In reality, whichever result can be computed first (because the requisite reads complete) is returned.

  If F_1 returns a value with label ℓ_1, and F_2 returns a value with label ℓ_2, $F_1 \vee F_2$ returns a value with label:

  $$(C(\ell_1) \sqcup C(\ell_2)) \quad (I(\ell_1) \sqcup I(\ell_2)) \quad (A(\ell_1) \sqcap A(\ell_2))$$

  The resulting value is more available than either F_1 or F_2's outputs alone. However, it can be influenced by either, and so it's only as trustworthy as their $\sqcup$. Likewise, it must be kept as secret as both, so confidentialities are joined as well.

In order to make use of these, our language must have some degree of functional composition.

I'm not sure how best to formalize this yet. One possible alternative that comes to mind is as follows.

Given a language of terminating functions f:

$$F := f \quad | \quad F \wedge F \quad | \quad F \vee F \quad | \quad F \circ_i F \quad | \quad remap\ F$$

Where $F_1 \circ_i F_2$ is functional composition, replacing the i^{th} input of the first function with the output of the second. If F_1 has n inputs, and F_2 has m, $F_1 \circ_i F_2$ has $m + n - 1$ inputs.

remap is a class of functions which take in m inputs, and assigns them (not necessarily in order, and some may be repeated or omitted) to the n inputs of F.

Power With this new, more expressive language of functions, we have message synthesizers, and can construct transactions that assemble data more available or trusted than any input. These can be used to construct a variety of complex protocols which we have examined in prior work [80].

3.5 Syntax

At this time, the syntax of TinyTX (meaning what the text formatting and keywords of this language will look like) has not yet been determined. Again, TinyTX is not meant to be used directly by programmers, but rather as an intermediate language for proving and reasoning about distributed programs.

One possible syntax would follow the semantics (Δs as a 4-tuple) quite closely. The name or id of the Δ would be at the beginning of a code block divided into 5 parts: one for each element of the tuple, with the functions computed separated into their own entry for simplicity.

```
NameHere {
  Reads { . . . }
  WriteSite { . . . }
  Functions { . . . }
  Writes { . . . }
  ChildTransactions{ . . . }
}
```

The first section would list a key to read on each line, along with some local variable name to represent the value read.

```
Reads {
  x := <key_site, key_label, key_identifier>
  y := <key_site, key_label, key_identifier>
  . . .
}
```

The second would simply name the site to which this SWST writes:

```
WriteSite{ Alice }
```

The third would assign (unique) variable names to the results of terminating functions which may take prior variable names (such as those from the Reads section) as inputs:

```
Functions {
  z := x + y
  w := z * y
  . . .
}
```

The fourth section would detail which variables now defined to write where:

```
Writes {
  x -> <Alice, key_label, key_identifier>
  w -> <Alice, key_label, key_identifier>
  . . .
}
```

Finally, the fifth section lists the names of child transactions to launch if certain variables now defined are true.

```
ChildTransactions {
  z => ThatOtherTransaction
  . . .
}
```

3.6 Implementation

No implementation for TinyTX has yet begun. We aim to create both an efficient runtime for TinyTX, and a compiler which will analyse whole programs (sets of SWSTs) to optimize performance. Ideally, the security properties of our type system and the correctness of our compiler would be formally verifiable in a system like Coq [62]. Certainly Coq's Galina specification language, which provides for provably terminating total functions, would be a strong candidate for the terminating functions in TinyTX.

4 Preliminary Results

This proposed thesis builds upon my previous work. It represents an alternative formalization which will hopefully provide easier / more automatic analysis of protocols such as Heterogeneous Fast Consensus, or Heterogeneous OARCast, which I have previously designed and checked painstakingly by hand [80]. This past research has provided insight into the vast and tragically underexplored space of distributed protocols with Heterogeneous Trust: any setting in which different participants have different ideas about which sites might fail and how.

For example, one previously analysed protocol, Heterogeneous Fast Consensus, generalizes the Bosco fast consensus algorithm. Without going into details of Consensus Algorithms, HFC customizes for particular data labels set by its participants. For some trust configurations of participants (constraints on who trusts data from whom), HFC can work in situations where no other Fast Consensus can. For one configuration of 5 participants, a traditional Fast Consensus such as Bosco would require 4 additional participants, and in simulation, takes almost twice as long to finish [80].

Prior work with the Fabric system [60] has revealed that complex protocols, and transactions especially, are tremendously difficult to get right. We found that Fabric's two-phase commit protocol could leak information about arbitrarily secret data through whether transactions involving those secrets aborted or not. If a participant not cleared for that secret were also participating in such a transaction, it could learn from abort messages whether there were conflicts at the secret.

By building complex protocols from one basic but powerful unit, the Single Write Site Transaction, I hope to simplify this analysis. For this reason, it is crucial that SWSTs and their implementation are secure, and the security rules about how they fit together are air-tight. However, once these are established, it is much easier to build and check complex systems. I believe it may provide a more sound foundation for work along those lines.

5 Work Plan

TinyTX will require 4 main steps:

- Complete a formal definition of TinyTX's syntax and semantics, complete with type and label definitions.

- Prove that this definition preserves Probabilistic Non-Interference.
- Implement an efficient and correct compiler and run-time system which can run TinyTX programs.
- Implement more complex, intricate protocols in TinyTX, and prove these formalizations are correct and secure.

I anticipate that the first two steps can be completed in a few months, depending on whether we attempt mechanically verifiable proofs of correctness. Ultimately, computer-checkable proofs are strongly desirable, but can prove extremely difficult to write.

Creating a correct and efficient compiler will take somewhat longer. I anticipate the project may require a year of work, if we want to implement existing transaction optimizations and efficient compilation. Our compiler will translate SWST descriptions to efficient code, likely in Java, C++, or a related language, and then take advantage of existing optimising compilers from there. Again, a computer-checkable proof that our compiler is correct is highly desirable. While some provably correct compilers exist ([54]), it is a daunting task, and will depend heavily on how complicated some optimizations are to verifiability prove correct. It also limits our available target languages to those with existing verified compilers, which may not include such attributes as network connections, critical to TinyTX.

6 Related Work

Programming languages for distributed systems (sometimes called *distributed programming languages*) are a topic of vigorous research. This work builds most closely on the Fabric project, which provides a rich languag with an information-flow type system, and persistent objects stored on remote, distributed stores [60]. Specifically, Fabric's type system is based on Jif's type sytem [67], which adapts the Decentralized Label Model [65, 18]. Much like in TinyTX, each piece of information or variable is labelled, and under no circumstances should the label of one piece influence any information not permitted by its label.

Fabric's high level approach has resulted in some tricky complexities. For example, each datum in Fabric has a security label limiting which other data it can affect, but it also needs another representing the security of *the fact*

that it is accessed. One might worry, for instance, that even though your regular plumber can't know the content of your message, the fact that you have sent a message to a rival plumber reveals information. As we have uncovered in our previous work on "abort channels," the mechanisms used to coordinate Fabric's transactions can themselves leak information. TinyTX seeks a much simpler approach: each datum has one label, we have only one core distributed mechanism to check, and the information flow properties within that mechanism are fairly straightforward. This requires us to make certain simplifying assumptions. For example, the security of the *fact that a read or write occurred* is exactly that of the key identifying value the read or written.

TinyTX is also able to express more complicated distributed protocols than Fabric, which operates entirely in 2-Phase-Commit-based transactions. Our core language assumes only that security labels form a lattice. (This has become a norm in the IF community since Denning [27].) However, an extension specifying Lantian Zheng's Confidentiality, Integrity, and Availability labels [99] would allow TinyTX to express and analyse the kind of intricate, heterogeneous-trust protocols that we have experience designing [80].

6.1 Security Languages

There is a wealth of prior research on security and information-flow typed languages, the vast majority of which are designed for single-site programs [73, 70, 41, 98, 74, 66, 9, 68, 19, 86, 28, 39]. As a distributed language, TinyTX is fundamentally different from most of these. However, TinyTX uses many of the techniques explored in prior languages. For example, we incorporate the label of a datum into the key under which it is stored. This avoids the classic "aliasing" problem, in which two different processes store data of different security levels in the same "place." This approach is strongly reminiscent of Secure Multi-Execution [28, 39], in which different copies of each Datum are kept for each security level. Our core label system, lattice, and sub-typing rules are all built on well-studied work [67, 18, 99].

Parallel programs have traditionally been difficult for security-typed languages [81, 87, 71, 94, 97]. Termination channels, or the ability to observe whether or not thread ever finishes, present one extremely difficult to analyse avenue of attack. We bypass at least part of this problem by requiring that all SWSTs be terminating: no correct program can run forever. It is still possible, with child transactions, that a kind of multi-SWST loop can exhibit

terminating or non-terminating behaviour. However, we are able to characterize which sites observe such behaviour, and can be certain that loops based on secret information only touch appropriate sites. We do not consider an adversary with the ability to see arbitrary network traffic.

Like TinyTX, many security-oriented languages maintain **non-interference** [63]. Loosely defined, non-interference requires that if two system states are indistinguishable to an observer before running a program, then after running the program on each system state, they remain indistinguishable from each other. This is tricky to adapt to a concurrent or parallel setting, since an observer might see changes made while the program is running. Two significant concurrent versions of non-interference emerge: possibilistic and probabilistic [81, 87, 94]. possibilistic non-interference holds that when running a program on two indistinguishable states, the set of possible outcomes (including all intermediate states visible to the observer) must be identical. Unfortunately, it may be the case that while the *set* of possible outcomes is the same for both states, the *odds* of some outcomes are different [81, 87]. Consider a program that guesses a random integer from 1 100, and if the value is exactly 1, outputs some secret integer, and otherwise outputs some random integer (chosen from the same range as the secret). Technically, any integer output is possible regardless of the secret, but it really seems that the secret is being leaked here. Probabilistic non-interference requires that the probability distribution over the possibility sets be equal as well.

One approach to probabilistic non-interference is *observational determinism* [94, 71], wherein information available to the observer before a program begins completely determines everything the observer can see. This disallows systems with true non-determinism, a staple of distributed systems [75]. Specifically, traditional distributed system design holds that two messages sent from different sites to the same site "at the same time" may be received in either order, non-deterministically [51]. Some approaches, such as the calculus $\lambda PARSEC$ [94], explicitly forbid such race-conditions. These approaches are impractical for modeling distributed systems where sites themselves are differently trusted: there is no way to avoid networks reordering messages. TinyTX instead allows non-deterministic scheduling, but restricts the set of sites which can influence scheduling to those allowed to influence information involved. Although non-deterministic, TinyTX embraces probabilistic non-interference. The only untrusted source of non-determinism which might violate non-interference is the network itself. Thwarting a net-

work adversary is surprisingly difficult, and the topic or orthogonal research on anonymous communication [24].

TinyTX does, however, ignore one covert channel of note: timing. Some sites may be able to observe, for example, how long an action which involved secret information took to complete. While this does not explicitly affect the computation they perform, it may leak information. Traditional distributed systems models assume the network is *asynchronous*, and it is impossible to place limits on the time networks will take to deliver messages, making it very difficult to avoid such timing channels [51, 75]. Some recent research suggests that these models may not be well suited to real-world systems, in which networks have more predictable properties [8]. There has been a great deal of research into mitigating the effectiveness of timing channels in secure programming languages under various assumptions [95, 73]. We leave the incorporation of such techniques into TinyTX to future work, as our core model is built along the lines of more traditional distributed systems models.

6.2 Distributed Languages

Language design for distributed systems is also a rich and active field of research [55, 37, 57, 30]. TinyTX is far from alone is seeking to provide a basis for systems with provable properties, security and otherwise [37, 60, 45, 11, 73, 89]. Many such languages are built for special purposes, such as data analystics or data-parallel computing, requiring specialized data structures and restricting what kinds of computation are possible [13, 3, 90, 50, 40, 91, 88]. Some are based around slow primitives, such as multi-phase commit procedures [60], or rely on processor-intense model-checking to prove properties [89].

Most distributed or concurrent languages are designed to make distributed computing easier or more intuitive for the programmer. This is an evident goal both in industrial languages like Erlang [4] and Go [38], and many research languages [37, 60, 13, 3, 90, 50, 40, 91, 88, 59, 92, 43, 11, 83, 15, 35, 45]. TinyTX does not share this objective. Instead, we seek to create a low-level language that is easy to analyse and compile to extremely efficient systems. Some pre-existing distributed languages with more complicated primitives may do well to compile *to* TinyTX.

TinyTX allows arbitrary data to be stored at arbitrary locations, specified by the program. Its primitives are comparatively small and light, while providing security and serializability, a very strong consistency guarantee [69].

TinyTX's primitives are also comparatively simple, making them easier to reason about without overlooking intricate corner cases. We hope that a language-based analysis of TinyTX's simple primitives will yield the ability to produce strong proofs about complex system properties without trying to build a complete model of all possible system states and checking all possible traces [89].

TinyTX has a close cousin in Zheng's DSR [97], which also presents a low-level distributed language with rich security labels. DSR, however, uses substantially different primitives, which make it more difficult to build up layers of abstraction (such as many sites working together to form an abstract site). DSR also is not designed around the idea of persistent storage, and was never implemented.

6.3 Distributed Frameworks

TinyTX is meant to provide low-level primitives for building secure distributed systems. Extremely popular frameworks like MapReduce [25], hadoop [23], and Dryad [44] have enjoyed widespread industrial popularity [53, 17], spawned a number of modifications and optimizations, and serve as the basis for several higher-level systems [1, 85, 29, 10, 33, 21]. While some of these modifications, notably Airavat [72] and SilverLine [49], add security properties to these frameworks, none account for cross-domain sites, which may be trusted with different secrets. While Airavat, for example, can provide differential privacy metrics (which TinyTX cannot), the labels used to express security policies on each datum are not nearly as flexible as TinyTX's.

TinyTX is designed to be a lower-level language than all of these frameworks. Indeed, MapReduce could itself be re-implemented in TinyTX. In this way, TinyTX is much less limiting in the kinds of computation it can express.

TinyTX's core mechanism, the Single Write-Site Transaction, is a lightweight primitive meant to provide **serializability**. Serializability is a popular abstraction that can help make distributed programs easier to reason about. It requires that for any observable system state, there exists some "serial" total ordering of transactions, such that if they each executed, one after another, from the beginning of the system, the observation would be explained. What's more, any later observation must have a serial ordering that begins with the one for the previous operation. This provides the illusion that transactions are executed individually, one after another [69]. This concept was

originally applied to transactions on databases (sometimes called "ACID"), but can be applied to distributed computing systems in general.

There are many existing distributed frameworks that build on low-level serializable transactions, including Sinfonia [2], Thor [58] Granola [22], Calvin [84], and ROCOCO [64]. Each of these ultimately requires a 2 round commit protocol for some transactions, making their basic components slower, but more expressive, than TinyTX's. Many of the optimization techniques developed for these systems can apply to, and should be used with, TinyTX. For example, ROCOCO makes heavy use of *transaction chopping* [78, 79, 96] to divide transactions into smaller atomic units, which can be safely committed independently. Granola identifies *independent distributed transactions*, based on data structures which do not require agreement on update orderings.

With the exception of Fabric [60], none of these kinds of frameworks offer the kind of language-based security policies TinyTX offers. Each however, represents an innovation in transaction processing, from which TinyTX can benefit. Database systems like IFDB [76] and various Multilevel Secure Databases [46, 82, 47, 6, 5, 48] also provide rich security policies, but make much stronger assumptions than TinyTX about how data is stored, and what computation can be performed.

Prior work on optimizing transactions will be critical to TinyTX's implementation. Work in coordination avoidance [7], commutative transactions [20, 56], convergent and commutative data types [77], and transaction chopping [14, 64] will be critical to ensuring the TinyTX compiler results in optimally efficient transactions. Each of these are techniques for identifying and removing cases of false-conflict: when two transactions need not be consistently ordered (the results would be the same for some interleaving), but ordering is required anyway.

TinyTX also borrows from some non-serializable systems. TinyTX's serialization mechanism is a modified version of traditional *causal consistency* mechanisms from systems like COPS [61], GentleRain [32], or Orbe [31]. Causal consistency requires that if some transaction A writes a value, and another transaction B reads that value, then A is scheduled before B. Unlike serializability, however, causal consistency does not require that the schedule be totally ordered: the *before* relation is only a strict partial order. This still allows for the possibility that multiple transactions happen "at the same time," (no one of them scheduled before any other) and so none read the results written by of any of the others. TinyTX's Single Write-Site Transactions (SWSTs), however, maintain true serializability, which is a strictly

stronger property.

7 Conclusion

Secure, analysable distributed systems have proven extremely difficult in the past. Nevertheless, they have become more important than ever, as medical records, personal communications, and surveillance data are all moving into massive, interconnected distributed systems. No existing language or framework is both general enough to express all distributed protocols, and possesses necessary security guarantees. TinyTX builds upon our prior work in distributed protocols and security languages, and will provide a low-level building block for complex, secure distributed systems. This will aid in the compilation and analysis of future distributed programs, and help programmers to avoid the mistakes of the past.

References

[1] Apache spark. http://spark.apache.org/, 2015.

[2] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *21st ACM Symp. on Operating System Principles (SOSP)*, pages 159–174, October 2007.

[3] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In *DATALOG*, 2010.

[4] Joe Armstrong. A history of erlang. In *HOPL*, 2007.

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

[6] Vijayalakshmi Atluri, Sushil Jajodia, Thomas F Keefe, Catherine D McCollum, and Ravi Mukkamala. Multilevel secure transaction processing: Status and prospects. *DBSec*, 8(1):79–98, 1996.

[7] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *PVLDB*, 8:185–196, 2014.

[8] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57:48–55, 2014.

[9] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 100–114, June 2004.

[10] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand S. Aiyer. Apache hadoop goes realtime at facebook. In *SIGMOD*, 2011.

[11] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

[12] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–240, 1985.

[13] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1:146–166, 1989.

[14] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Transaction chopping for parallel snapshot isolation. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 388–404. Springer Berlin Heidelberg, 2015.

[15] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21:291–312, 2007.

[16] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984.

[17] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *CoRR*, abs/1208.4174, 2012.

[18] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 242–253, July 2006.

[19] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2009.

[20] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*, 2013.

[21] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, 2010.

[22] James A. Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX*, 2012.

[23] Doug Cutting and Mike Cafarella. hadoop. Software release, `https://hadoop.apache.org/`, January 2016.

[24] George Danezis and Claudia Diaz. A survey of anonymous communication channels. 2008.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[26] Dorothy E. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, W. Lafayette, Indiana, USA, May 1975.

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

[28] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, May 2010.

[29] Jens Dittrich, Jorge-Arnulfo Quiane-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jorg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3:518–529, 2010.

[30] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, 2016.

[31] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In *CLOUD*, 2013.

[32] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *CLOUD*, 2014.

[33] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, 2010.

[34] 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, November 1976. Also published as IBM RJ1487, December, 1974.

[35] Matthew Fluet, Mike Rainey, John H. Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *POPL*, 2007.

[36] Simon N. Foley. A taxonomy for information flow policies and models. In *IEEE Symp. on Security and Privacy*, pages 98–108, 1991.

[37] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In *POPL*, 2016.

[38] Robert Griesemer, Rob Pike, and Ken Thompson. The Go Programming Language. Software release, `https://golang.org/`, November 2015.

[39] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *CCS*, 2012.

[40] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.

[41] William R. Harris, Somesh Jha, Thomas W. Reps, Jonathan Anderson, and Robert N. M. Watson. Declarative, temporal, and practical programming with capabilities. In *SP*, 2013.

[42] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In Gert Smolka, editor, *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer Berlin Heidelberg, 2000.

[43] Parry Husbands, Costin Iancu, and Katherine A. Yelick. A performance analysis of the berkeley upc compiler. In *ICS*, 2003.

[44] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EUROSYS*, 2007.

[45] Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

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

[47] I.E. Kang and T.F. Keefe. On transaction processing for multilevel secure replicated databases. In Yves Deswarte, Grard Eizenberg, and Jean-Jacques Quisquater, editors, *Computer Security ESORICS 92*, volume 648 of *Lecture Notes in Computer Science*, pages 329–347. Springer Berlin Heidelberg, 1992.

[48] Navdeep Kaur, Rajwinder Singh, Manoj Misra, and Anil Kumar Sarje. Concurrency control for multilevel secure databases. *I. J. Network Security*, 9:70–81, 2009.

[49] Safwan Mahmud Khan, Kevin W. Hamlen, and Murat Kantarcioglu. Silver lining: Enforcing secure information flow at the cloud edge. In *ic2e*, 2014.

[50] Arvind Krishnamurthy, Alexander Aiken, Phillip Colella, David Gay, Susan L. Graham, Paul N. Hilfinger, Ben Liblit, Carleton Miyamoto,

Geoff Pike, Luigi Semenzato, and Katherine A. Yelick. Titanium: A high performance java dialect. In *PPSC*, 1998.

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

[52] L. Lamport. The part-time parliament. Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[53] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *SIGMOD Record*, 40:11–20, 2011.

[54] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, 2009.

[55] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, 2016.

[56] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguia, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10^{th} USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2012.

[57] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *POPL*, 2016.

[58] B. Liskov. Thor: An object-oriented database system, 1991.

[59] B. Liskov and R. W. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. on Programming Languages and Systems*, 5(3):381–404, July 1983.

[60] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22^{nd} ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.

[61] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for

wide-area storage with COPS. In *23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.

[62] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

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

[64] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.

[65] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *16th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, October 1997.

[66] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[67] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[68] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symp. on Security and Privacy*, pages 165–179, 2011.

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

[70] François Pottier and Vincent Simonet. Information flow inference for ML. In *29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[71] A. W. Roscoe. CSP and determinism in security modelling. In *IEEE Symp. on Security and Privacy*, pages 114–127, May 1995.

[72] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, 2010.

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

[74] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 255–269, June 2005.

[75] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[76] David A. Schultz and Barbara Liskov. Ifdb: decentralized information flow control for databases. In *EUROSYS*, 2013.

[77] Marc Shapiro, Nuno M. Preguica, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.

[78] Dennis Shasha, Franois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20:325–363, 1995.

[79] Dennis Shasha, Eric Simon, and Patrick Valduriez. Simple rational guidance for chopping up transactions. In *SIGMOD*, 1992.

[80] Isaac C. Sheff, Robbert van Renesse, and Andrew C. Myers. Distributed protocols and heterogeneous trust: Technical report. Technical Report arXiv:1412.3136, Cornell University Computer and Information Science, 2014.

[81] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, January 1998.

[82] K.P. Smith, B.T. 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.

[83] Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. Aeminium: a permission

based concurrent-by-default programming language approach. In *PLDI*, 2014.

[84] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[85] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[86] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *7th International Joint Conference on the Theory and Practice of Software Development*, pages 607–621, 1997.

[87] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2,3):231–253, November 1999.

[88] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian T. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37:633–652, 2011.

[89] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41, 2009.

[90] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.

[91] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[92] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HOTCLOUD*, 2010.

[93] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In David Sands, editor, *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61. Springer Berlin Heidelberg, 2001.

[94] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 29–43, June 2003.

[95] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, June 2012.

[96] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.

[97] Lantian Zheng. *Making distributed computation secure by construction.* PhD thesis, Cornell University, Ithaca, New York, USA, January 2007.

[98] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, August 2004.

[99] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 272–286, June 2005.