On the Flow of Data, Information, and Time

Martín Abadi[1] and Michael Isard[2]

[1] University of California, Santa Cruz, California, USA
[2] Microsoft Research*, Mountain View, California, USA

Abstract. We study information flow in a model for data-parallel computing. We show how an extant notion of virtual time can help guarantee information-flow properties. For this purpose, we introduce functions that express dependencies between inputs and outputs at each node in a dataflow graph. Each node may operate over a distinct set of virtual times—so, from a security perspective, it may have its own classification scheme. A coherence criterion ensures that those local dependencies yield global properties.

1 Introduction

The flow of data generally entails the flow of information, whose understanding is often essential for the performance and correctness of dataflow computations. For example, knowing that two dataflow computations on different input batches do not interfere with one another can open opportunities for asynchronous, overlapped execution. It may perhaps also contribute to ensuring that sensitive inputs do not leak through public outputs, that untrusted data does not taint trusted results, and other security and privacy properties.

Therefore, modern platforms for data-parallel computing sometimes track dependencies, at least coarsely, primarily in order to enable efficient implementations. For instance, Spark maintains dependencies between Resilient Distributed Datasets [17], representing their lineage. Naiad [11] associates messages and other events with virtual times [4]; the partial order of virtual times, which need not correspond to the order of execution, determines whether one event can potentially result in another event.

Of course, understanding the flow of information does not necessarily mean the same in data-parallel computing and in security and privacy. In particular, covert communication channels are seldom a concern for data-parallel computing. Furthermore, at least at present, systems for data-parallel computing typically leverage strong trust assumptions: most systems code is trusted, and even the environment is often assumed to be somewhat benign.

Nevertheless, we explore the idea that models and systems for data-parallel computing can offer substantial information-flow control. We focus on concepts and facilities for information-flow control, rather than on their applications. Specifically, we consider the computational model that underlies Naiad, called

* Most of this work was done at Microsoft Research. M. Abadi is now at Google.

R. Focardi and A. Myers (Eds.): POST 2015, LNCS 9036, pp. 73–92, 2015.
DOI: 10.1007/978-3-662-46666-7_5

timely dataflow. We find that, after a modest strengthening (and a change of perspective), timely dataflow offers information-flow properties that resemble familiar ones from the security literature.

As indicated above, timely dataflow supports partially ordered virtual times. These virtual times may be viewed as analogous to security levels or classifications. Furthermore, timely dataflow considers the question of whether one event at a given virtual time t and location l in a dataflow graph could result in another event at a virtual time t' and location l' in the same graph. The expectation that an event at (l, t) cannot result in an event at (l', t') "in the past" is analogous to conditions on flows across security levels, but weaker. So we identify alternative concepts and properties that, although consistent with timely dataflow, lead to non-interference guarantees.

One somewhat unusual aspect of the resulting framework is that it allows the use of different sets of virtual times (that is, different sets of security levels) in different parts of a system. For example, virtual times inside loops may have coordinates that correspond to loop counters, and can distinguish data from different loop iterations that may be processed simultaneously; those coordinates do not make sense outside loops. From a security viewpoint, virtual times in different parts of a computation may reflect the classification schemes of different organizations, or the classification schemes appropriate to the different kinds of data being processed. While simple levels like "Public" and "Secret" are allowed, there is no built-in assumption or requirement that they mean and are treated the same everywhere. Moreover, each neighborhood of a dataflow graph could have its own custom levels. Finally, a virtual time may be a tuple that includes both structural information (such as loop counters) and other facets, such as secrecy and integrity levels. We define a criterion that ensures the coherence of the use of levels.

Our main results enable us to reason about systems organized as dataflow graphs, and to characterize the information that each node in such a graph may obtain. As a small example (to which we return in Section 5.4), consider a system that receives and processes messages that each pertain to one of two users U_1 and U_2. Suppose that a particular node p_0 in this system forwards data about each user to a different destination, p_1 or p_2 respectively. We abstract p_0's behavior by stating that its messages to p_1 do not depend on its inputs about U_2, and symmetrically its messages to p_2 do not depend on its inputs about U_1. We make such statements directly, formulating them in terms of virtual times; in other approaches, analogous statements might be encoded in type annotations. From p_0's properties and the topology of the dataflow graph we may then derive that p_1 learns nothing from the inputs about U_2, and that p_2 learns nothing from the inputs about U_1. More generally, our work provides an approach for establishing information-flow properties of a dataflow system from properties of individual nodes and the topology of the system.

The next section is a review of the relevant aspects of the model of computation that we consider. Section 3 introduces auxiliary concepts: frontiers, filtering, and reordering. Section 4 defines and studies the machinery for specifying

dependency information at the level of individual nodes. Section 5 presents lemmas and our main results, including the coherence criterion and the non-interference guarantees. Section 6 concludes. Although this paper aims to be self-contained, it stems from a larger effort to understand, improve, and apply timely dataflow. Section 6 briefly discusses aspects of this effort relevant to security and some directions for further work. Because of space constraints, proofs are omitted.

2 Model of Computation

This section reviews the setting for our work. As explained in Section 6, it is a fragment of the full timely dataflow model, which was introduced in the context of Naiad [11] and whose formal study is the subject of another paper, currently in preparation. Here, therefore, we do not describe the model in full detail, focusing instead on the main ideas and aspects relevant to our present purposes.

As in other dataflow models (e.g., [5]), programs are organized as directed graphs, in which nodes do the processing and messages travel on edges. We write P for the set of nodes (or processors) and E for the set of edges (or channels). We refer to both nodes and edges as locations. For simplicity, we assume that the source $src(e)$ and the destination $dst(e)$ of each edge e are distinct nodes; however, in general, graphs may contain cycles. We write M for the set of messages, and M^* for the set of finite sequences of messages.

Each message m is associated with a virtual time $time(m)$. The virtual times form a partial order (not necessarily linear, not necessarily a lattice), which we write $(T, \leq)$. There is no built-in requirement that the order of processing of messages correspond in any way to their virtual times.

We can describe the state of a system as a mapping from nodes to their local states plus a mapping from edges to their contents. We write $LocState(p)$ for the local state of node p, and Σ_{Loc} for the set of local states; we are not concerned with the specifics of how local state is organized. We write $Q(e)$ for the finite sequence of messages on edge e.

A local history for a node p is a finite sequence $\langle\!\langle s, (e_1, m_1), \ldots, (e_k, m_k) \rangle\!\rangle$ that starts with an initial local state s that satisfies a given predicate $Initial(p)$, and is followed by (zero, one, or more) pairs of the form (e_i, m_i), which indicate the messages that the node has received and the corresponding edges. We write $Histories(p)$ for the set of local histories of p.

We assume that initially each node p is in a local state that satisfies $Initial(p)$, and for each edge e we let $Q(e)$ contain an arbitrary finite sequence of messages, so as to get computations started. (This detail constitutes a minor variation from other presentations of timely dataflow, in which computations can get started by other means.) Thereafter, at each step of computation (atomically, for simplicity), a node that has messages on incoming edges picks one of them, processes it, and places messages on its output edges. The processing is defined by a function $g_1(p)$ for each node p, which we apply to p's local state s and to a pair (e, m),

and which produces a tuple that contains a new state s' and finite sequences of messages $\mu_1, \ldots, \mu_k$ on p's output edges $e_1, \ldots, e_k$, respectively. We write:

$$g_1(p)(s, (e, m)) = (s', \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$$

where $\langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle$ is the function that maps e_1 to μ_1, ..., e_k to μ_k. Iterating this function $g_1(p)$, we obtain a function $g(p)$ which takes as input an entire local history h and produces a new state s' and the cumulative finite sequences of messages $\mu_1, \ldots, \mu_k$ for the output edges $e_1, \ldots, e_k$, as follows:

- $g(p)(\langle\langle s \rangle\rangle) = (s, \langle e_1 \mapsto \emptyset, \ldots, e_k \mapsto \emptyset \rangle)$,
- if $g(p)(h) = (s', \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle)$ and $g_1(p)(s', (d, m)) = (s", \langle e_1 \mapsto \mu'_1, \ldots, e_k \mapsto \mu'_k \rangle)$, then $g(p)(h \cdot (d, m)) = (s", \langle e_1 \mapsto \mu_1 \cdot \mu'_1, \ldots, e_k \mapsto \mu_k \cdot \mu'_k \rangle)$.

As in this definition, we write $\emptyset$ for the empty sequence and $\langle\langle a_0, a_1, \ldots \rangle\rangle$ for a sequence that contains a_0, a_1, ..., and we use $\cdot$ both for adding elements to sequences and for appending sequences. We let $\Pi_{\mathrm{Loc}}(s', \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle) = s'$ and $\Pi_{e_i}(s', \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle) = \mu_i$ for $i = 1 \ldots k$.

The overall specification of a system denotes a set of allowed sequences of states. Each of the sequences starts in an initial state, and every pair of consecutive states is either identical (a "stutter") or related by a step of computation. We add an auxiliary state function H (a history variable [1]) in order to track local histories: $H(p)$ represents p's local history; thus, each state is defined by values for the state functions *LocState*, Q, and H. We express the specification in TLA [8], in Figure 1, with the following notations. A primed state function (Q', *LocState*$'$, or H') in an action refers to the value of the state function in the "next" state (the state after the action); $\Box$ is the temporal-logic operator "always"; given an action N and a list of expressions $v_1, \ldots, v_k$, $[N]_{v_1,\ldots,v_k}$ abbreviates $N \vee ((v'_1 = v_1) \wedge \ldots \wedge (v'_k = v_k))$.

We call *ISpec* the complete specification, *InitProp* the initial conditions, and *MessR* the action that represents a step of computation. When Q_0 is a (state-independent) function from E to M^*, we also write *ISpec*(Q_0) for the conjunction of *ISpec* with $\forall e \in E.Q(e) = Q_0(e)$, which says that the initial values of the queues are as given by Q_0.

The definition of the action *MessR* describes how a node p dequeues a message m and reacts to it, producing messages. This action is a relaxed version of a simpler action that we call *Mess* (hence the name *MessR*) and according to which p takes a message from the head of $Q(e)$, so $Q(e) = m \cdot Q'(e)$. (The head of a queue is to the left, the tail to the right.) According to *MessR*, on the other hand, p is allowed to take any message m in $Q(e)$ such that there is no message n ahead of m with $time(n) \leq time(m)$; so, for some u and v, $Q(e) = u \cdot m \cdot v$, $Q'(e) = u \cdot v$, and u does not contain any message n with $time(n) \leq time(m)$. Thus, queues are not strictly FIFO. This relaxation can be useful in support of optimizations, as it can allow more messages for a given time to be processed together. It is

also important for work on fault-tolerance in which we are currently engaged, and seems attractive in the present context as well. (See Section 5.)

$$InitProp \triangleq \left(\begin{array}{l} \forall p \in P.LocState(p) \in Initial(p) \\ \wedge \\ \forall e \in E.Q(e) \in M^* \\ \wedge \\ \forall p \in P.H(p) = \langle\!\langle LocState(p) \rangle\!\rangle \end{array}\right)$$

$$MessR \triangleq \exists p \in P.MessR1(p)$$

$$MessR1(p) \triangleq \left(\begin{array}{l} \exists m \in M. \exists e \in E \text{ such that } p = dst(e). \exists u, v \in M^*. \\ Q(e) = u{\cdot}m{\cdot}v \wedge Q'(e) = u{\cdot}v \\ \wedge \\ \forall n \in u.time(n) \not\leq time(m) \\ \wedge \\ Mess2(p, e, m) \end{array}\right)$$

$$Mess2(p, e, m) \triangleq \left(\begin{array}{l} let \\ \{e_1, \ldots, e_k\} = \{d \in E \mid src(d) = p\}, \\ s = LocState(p), \\ (s', \langle e_1 \mapsto \mu_1, \ldots, e_k \mapsto \mu_k \rangle) = g_1(p)(s, (e, m)) \\ in \\ LocState'(p) = s' \\ \wedge \\ Q'(e_1) = Q(e_1){\cdot}\mu_1 \ldots Q'(e_k) = Q(e_k){\cdot}\mu_k \\ \wedge \\ H'(p) = H(p){\cdot}(e, m) \\ \wedge \\ \forall q \in P \neq p.LocState'(q) = LocState(q) \\ \wedge \\ \forall d \in E - \{e, e_1, \ldots, e_k\}.Q'(d) = Q(d) \\ \wedge \\ \forall q \in P \neq p.H'(q) = H(q) \end{array}\right)$$

$$ISpec \triangleq InitProp \wedge \Box[MessR]_{LocState,Q,H}$$

$$ISpec(Q_0) = ISpec \wedge \forall e \in E.Q(e) = Q_0(e)$$

Fig. 1. The specification

3 Frontiers, Filtering, and Other Auxiliary Concepts

We introduce a few auxiliary notions, namely frontiers, filtering, and reordering.

3.1 Frontiers

A subset S of T is *downward closed* if and only if, for all t and t', $t \in S$ and $t' \leq t$ imply $t' \in S$. We call such a subset a *frontier*, and write F for the set of frontiers; we often let f range over frontiers. When $S \subseteq T$, we write $Close_{\downarrow}(S)$ for the downward closure of S (the least frontier that contains S).

As indicated in the Introduction, we may view virtual times as security levels. From that perspective, a frontier is a set of security levels S such that if S includes one level it includes all lower levels. For example, in multi-level security (MLS), such a set S might arise as the set of levels of the objects that a subject at a given level can read.

3.2 Filtering

We introduce *filtering* operations on histories and on sequences of messages. These filtering operations keep or remove all elements whose times are in a given frontier. Thus, they are analogous to the *purge* functions that appear in security models. (See, for example, McLean's survey [9, Section 2.2.1].)

Given a local history h and a frontier f, we write $h@f$ for the subsequence of h obtained by removing (filtering out) all events (d, m) such that $time(m) \notin f$. More precisely, $h@f$ is defined inductively by:

- $\langle\!\langle s \rangle\!\rangle @f = \langle\!\langle s \rangle\!\rangle$,
- $(h \cdot (d, m))@f = (h@f) \cdot (d, m)$ if $time(m) \in f$ and $(h \cdot (d, m))@f = h@f$ otherwise.

Similarly, when u is a sequence of messages, we write $u@f$ for the subsequence obtained by removing those messages whose times are not in f. Finally, given a sequence of messages u and a frontier f, we write $u@\!\!\!\!\setminus f$ for the subsequence of u consisting only of messages whose times are not in f.

3.3 Reordering

We define a relation $\hookrightarrow$ on finite sequences of messages: it is the least reflexive and transitive relation such that, for $u, v \in M^*$ and $m_1, m_2 \in M$, if $time(m_1) \not\leq time(m_2)$ then $u \cdot m_1 \cdot m_2 \cdot v \hookrightarrow u \cdot m_2 \cdot m_1 \cdot v$. We call it the *reordering* relation.

This relation is a counterpart at the level of message sequences to the reordering that happens in message processing according to action *MessR* of Figure 1. It is therefore helpful for analyzing that specification and its implementations.

3.4 Subtraction

Subtraction for message sequences $(-)$ is defined inductively by:

$$\begin{aligned} u - \emptyset &= u \\ u - m{\cdot}v &= (u - m) - v \\ (m{\cdot}u - m) &= u \\ (m'{\cdot}u - m) &= m'{\cdot}(u - m) \text{ for } m' \neq m \\ \emptyset - m &= \emptyset \end{aligned}$$

The last clause $(\emptyset - m = \emptyset)$ appears in order to make subtraction a total operation. In our uses of subtraction, we sometimes ensure explicitly that it does not apply.

3.5 Some Properties of Filtering and Reordering

We state a few properties of filtering and reordering. Throughout, f, f_1, f_2, f_3 range over frontiers; u, v, w range over message sequences; h ranges over histories.

Proposition 1. *If $f_1 = f_2 \cap f_3$ then $h@f_1 = h@f_2@f_3$.*

Proposition 2. *If $u \hookrightarrow v$ then $u@f \hookrightarrow v@f$.*

Proposition 3. *If $u \hookrightarrow v$ then $(u - w) \hookrightarrow (v - w)$.*

Proposition 4. *If $u \hookrightarrow v{\cdot}w$ then $(u - v@f) \hookrightarrow v@f{\cdot}w$.*

Proposition 5. *If $u@f = u'@f$, then $(u - v@f)@f = (u' - v@f)@f$.*

The last two propositions (Propositions 6 and 7) are useful in reasoning with the action *MessR* because they provide methods for establishing that, for a sequence u and element m, there is no element n with $time(n) \leq time(m)$ to the left of m in u (in a prefix v). They say, respectively, that it suffices to consider any reordering u' of u or any sequence u' that coincides with u on some frontier f such that $time(m) \in f$.

Proposition 6. *If $u \hookrightarrow u'$, $u' = v'{\cdot}m{\cdot}w'$, and $time(n) \not\leq time(m)$ for all n in v', then there exist v and w such that $u = v{\cdot}m{\cdot}w$, and $time(n) \not\leq time(m)$ for all n in v.*

Proposition 7. *If $u@f = u'@f$, $u' = v'{\cdot}m{\cdot}w'$, $time(m) \in f$, and $time(n) \not\leq time(m)$ for all n in v', then there exist v and w such that $u = v{\cdot}m{\cdot}w$, and $time(n) \not\leq time(m)$ for all n in v.*

4 From Timeliness to Determination

Time domains and the could-result-in relation are central to timely dataflow. Although we do not need a formal definition of these notions for our present purposes, we review them informally in this section, in order to motivate our new definitions. While the could-result-in relation focuses on whether one event might trigger another event (directly or indirectly), we are interested in whether a history or a part of a history suffices for determining an output. These two questions are closely related, as we show. We treat the latter via *frontier transformers*, which are functions that map frontiers for inputs to frontiers for outputs and which we introduce and study in this section.

4.1 Time Domains

Timely dataflow does not require that all nodes deal with the same set of virtual times. In particular, the set T may be the disjoint union of multiple sets T_p, which we call *time domains*, one for each node p in a dataflow graph. Node p may expect inputs with times in set T_p and produce outputs with times in the sets appropriate for their recipients.

For example, in Naiad, nodes for loop ingress expect inputs with times of the form $(t_1, \ldots, t_k)$, and produce outputs with an extra coordinate, set to 0: $(t_1, \ldots, t_k, 0)$. Nodes for loop egress expect inputs with times of the form $(t_1, \ldots, t_k, t_{k+1})$, and drop the last coordinate on outputs. Nodes for loop feedback expect inputs with times of the form $(t_1, \ldots, t_k)$, and increment the last coordinate of these times. In all cases, the appropriate value of k is determined by the nesting depth of the loop.

Beyond these standard examples, it is possible, at least in principle, for programmers to define custom nodes, with their own ideas about virtual times. Thus, a custom node may consume inputs with times 1 and 2, but, somehow, produce results with times "Public" and "Secret"; or a custom node may consume inputs with times "Public" and "Secret", but produce results with finer classifications, such as "(Secret,A)" or "(Secret,B)", where "A" and "B" might indicate compartments, retention policies, or other properties of interest.

For simplicity, we proceed with the assumption that all inputs of a node are in the same time domain, but the outputs on each outgoing edge may be in a different time domain. It is straightforward to accommodate inputs in different time domains by inserting relay nodes that translate across time domains on incoming edges.

4.2 The Could-result-in Relation

When one event at a given virtual time t and location l in a dataflow graph can potentially result in another event at a virtual time t' and location l' in the same graph, we say that (l, t) could-result-in (l', t'). We write this relation $(l, t) \rightsquigarrow (l', t')$. For example, suppose that whenever node p receives any message m with $time(m) = 1$ on incoming edge d, p outputs a message n with $time(n) = 2$

on outgoing edge e; in this case we would have that $(d, 1) \rightsquigarrow (e, 2)$. In Naiad, the could-result-in relation is exploited for supporting completion notifications, which tell a node when it will no longer see messages for a given time. It also allows an implementation to reclaim resources that correspond to pairs (l, t) at which no more events are possible.

Informally, we expect that an event at (l, t) cannot result in an event at (l', t') "in the past". Naiad relies on this property in some of its algorithms. It holds rather obviously for most nodes, since, in response to an input at time t, most nodes would produce outputs at the same time t. However, defining "in the past" is delicate across time domains; fortunately, the approach that we develop in this paper does not require it.

As suggested in the Introduction, the expectation that an event cannot result in another event "in the past" is somewhat analogous to conditions on flows across security levels. For example, one may generally expect that a "low-integrity" event cannot cause a "high-integrity" event, except perhaps in trusted system components. Obviously, however, this property is not quite equivalent to a non-interference guarantee, or to other strong guarantees defined in the security literature [9]. Even if an input on edge d at time 2 may not trigger an output on edge e at time 1 for a node p, so we do not have $(d, 2) \rightsquigarrow (e, 1)$, the input at time 2 may affect the contents of future messages at time 1, if p is stateful and sends such messages in response to future inputs at times 0 and 1. Thus, inputs at time 2 may interfere with outputs at time 1.

4.3 Frontier Transformers

Going beyond what the could-result-in relation can express, knowing whether subsets of inputs determine subsets of outputs can be useful for a variety of purposes. We are finding it valuable in the context of current work on fault-tolerance. It is also clearly valuable for security, in which we often want, for instance, that "Public" inputs determine "Public" outputs, or that "Trusted" inputs determine "Trusted" outputs.

Formally, for each edge $e \in E$, we assume a function $\phi(e)$ that maps frontiers to frontiers (so, $\phi(e)$ is a frontier transformer). Its main intended property is Condition 1 which says that h gives rise to a message on e in $\phi(e)(f)$ if and only if so does $h@f$, and with messages in the same order and multiplicity.

Condition 1. *For all $f \in F$, if $g(p)(h) = (\ldots, \langle \ldots e_i \mapsto \mu_i \ldots \rangle)$ and $g(p)(h@f) = (\ldots, \langle \ldots e_i \mapsto \mu'_i \ldots \rangle)$ then $\mu_i @ \phi(e_i)(f) = \mu'_i @ \phi(e_i)(f)$.*

For many simple nodes, $\phi(e)$ may be the identity function for all outgoing edges e. On the other hand, the identity function is not always appropriate, particularly (but not only) when a node produces outputs in a different time domain than its inputs. Some of the nodes described in Section 4.1 exemplify this point. Entering a loop at depth $k+1$, inputs to an ingress node in a frontier f determine outputs for all times $\{(t_1, \ldots, t_k, t_{k+1}) \mid (t_1, \ldots, t_k) \in f\}$. In a loop at depth k, inputs to a feedback node in a frontier f determine outputs in

$\{(t_1, \ldots, t_k + 1) \mid (t_1, \ldots, t_k) \in f\}$. As another simple example, when T consists of two unrelated points t_1 and t_2 that represent private data for two users U_1 and U_2, we may have a node with outgoing edges e_1 and e_2 that demultiplexes data for U_1 and U_2, so that $\phi(e_1)(\{t_1\}) = T$ and $\phi(e_2)(\{t_2\}) = T$.

The function ϕ need not be as accurate as possible. In particular, $\phi(e)$ could always be completely uninformative (as small as possible), with $\phi(e)(f) = \emptyset$ for all $f \neq T$ and $\phi(e)(T) = T$. However, a more informative ϕ is typically more helpful, and generally easy to find.

In this paper, we do not investigate how to check that a node actually satisfies Condition 1 for a given ϕ. Section 6 returns briefly to this subject.

4.4 Relating ϕ to $\rightsquigarrow$

With the aim of clarifying the relation between ϕ and $\rightsquigarrow$, we argue that $\rightsquigarrow$ is included in ϕ at each node. More precisely, if an event at a node p at time t_1 could-result-in an event at time t_2 on one of the outgoing edges e, and t_2 is in $\phi(e)(f)$ for some frontier f, then t_1 is in f. For example, if f includes only the security level "Public", and $\phi(e)$ is simply the identity function, this property entails that if an event at p at time t_1 could-result-in a message on e at the level "Public", then t_1 is also in f and hence equals "Public".

Proposition 8. *Assume that ϕ satisfies Condition 1. Suppose $src(e) = p$ and $(p, t_1) \rightsquigarrow (e, t_2)$. Then, for all f, if $t_2 \in \phi(e)(f)$ then $t_1 \in f$.*

This proposition relies on the following property of $\rightsquigarrow$: if $(p, t_1) \rightsquigarrow (e, t_2)$ and $src(e) = p$ then there exist a history h for p, a state s such that

$$g(p)(h) = (s, \ldots)$$

and an event (d, m) such that $t_1 \leq time(m)$ and

$$g_1(p)(s, (d, m)) = (\ldots, \langle \ldots e \mapsto \mu \ldots \rangle)$$

where some element of μ has time $\leq t_2$. In this paper we simply assume this property; the proof that it actually holds requires a definition of $\rightsquigarrow$, which we omit.

4.5 A Special Case of Condition 1

In the security literature, non-interference properties are sometimes expressed in terms of single levels (e.g., outputs at level "Trusted" are determined by inputs at level "Trusted", or outputs to a user U are determined by U's inputs), rather than in terms of sets of levels analogous to frontiers. McLean's survey [9], for example, phrases purging functions and non-interference in terms of individual users, while the classic article by Goguen and Meseguer [3] refers to groups of users.

We therefore investigate the power of a special case of Condition 1 in which the frontier f is not arbitrary but rather consists of (the downward closure of) a

single time. Such a special case is often sufficient, and sometimes equivalent to the full Condition 1. In particular, when $(T, \leq)$ is a finite linear order, the only frontiers are $\emptyset$ and the sets of the form $Close_{\downarrow}(\{t\})$ for some $t \in T$.

Condition 2 captures this special case. It specializes Condition 1 to f of the form $Close_{\downarrow}(\{t\})$, for $t \in T$. It does not require that $\phi(e_i)(f)$ be of the same form.

Condition 2. *For all $t \in T$, if $f = Close_{\downarrow}(\{t\})$, $g(p)(h) = (\ldots, \langle \ldots e_i \mapsto \mu_i \ldots \rangle)$, and $g(p)(h@f) = (\ldots, \langle \ldots e_i \mapsto \mu'_i \ldots \rangle)$ then $\mu_i @ \phi(e_i)(f) = \mu'_i @ \phi(e_i)(f)$.*

We generally adopt Condition 1 rather than Condition 2, because Condition 2 is strictly weaker than Condition 1. The following small but tricky example illustrates this point. Perhaps with the security literature in mind (e.g., [2]), one may imagine that a lattice structure for the set of times T would help, and specifically that it would enable us to represent an arbitrary frontier f by the least upper bound of its elements. However, a variant of the example shows that Condition 2 is strictly weaker than Condition 1 even if T is a very simple distributive lattice.

Example 1. Suppose that T consists of three unrelated elements a, b, and c, and a fourth element d below b and c but not a.

The example concerns a simple node p that ignores its initial state. It has a single input edge e and a single output edge e', for which we take $\phi(e')(f) = f$. Moreover, the node ignores the contents of input messages, considering only their times. It also ignores all input messages at times b and c. As output, it may produce $\emptyset$, $\langle\!\langle m_b, m_c \rangle\!\rangle$, or $\langle\!\langle m_c, m_b \rangle\!\rangle$, where m_b and m_c are distinct, fixed messages with $time(m_b) = b$ and $time(m_c) = c$. So the function $g(p)$ for this node can be regarded as mapping a sequence of (a and d) times for input messages to $\emptyset$, $\langle\!\langle m_b, m_c \rangle\!\rangle$, or $\langle\!\langle m_c, m_b \rangle\!\rangle$. We write $\bar{g}$ for this mapping, and define it as follows:

$$\begin{aligned} \bar{g}(a^*) &= \emptyset \\ \bar{g}(a^+ \cdot d \cdot u) &= \langle\!\langle m_b, m_c \rangle\!\rangle \\ \bar{g}(d \cdot u) &= \langle\!\langle m_c, m_b \rangle\!\rangle \end{aligned}$$

where u is an arbitrary sequence of a's and d's. It is straightforward to define a function $g_1(p)$ that induces a function $g(p)$ that corresponds to $\bar{g}$.

Let $f = \{b, c, d\}$. Condition 1 fails for this f. We have that $\bar{g}((a \cdot d)@f) = \bar{g}(d) = \langle\!\langle m_c, m_b \rangle\!\rangle$, so $\bar{g}((a \cdot d)@f)@f = \langle\!\langle m_c, m_b \rangle\!\rangle$, while $\bar{g}(a \cdot d) = \langle\!\langle m_b, m_c \rangle\!\rangle$, so $\bar{g}(a \cdot d)@f = \langle\!\langle m_b, m_c \rangle\!\rangle$, hence

$$\bar{g}((a \cdot d)@f)@f \neq \bar{g}(a \cdot d)@f$$

On the other hand, in the special case of frontiers of the form $Close_{\downarrow}(\{t\})$, where $t \in T$, Condition 1 holds:

- For $t = a$: For all u, $\bar{g}(u@Close_{\downarrow}(\{a\})) = \emptyset$, and $\bar{g}(u)$ never contains a message at time a, so

$$\bar{g}(u@Close_{\downarrow}(\{a\}))@Close_{\downarrow}(\{a\}) = \bar{g}(u)@Close_{\downarrow}(\{a\})$$

– For $t = b$: For all u, $\bar{g}(u@Close_{\downarrow}(\{b\})) = \langle\!\langle m_c, m_b \rangle\!\rangle$ if u contains a d, and is $\emptyset$ otherwise; so $\bar{g}(u@Close_{\downarrow}(\{b\}))@Close_{\downarrow}(\{b\}) = \langle\!\langle m_b \rangle\!\rangle$ if u contains a d, and is $\emptyset$ otherwise. On the other hand, $\bar{g}(u) = \langle\!\langle m_c, m_b \rangle\!\rangle$ or $\langle\!\langle m_b, m_c \rangle\!\rangle$ if u contains a d, and is $\emptyset$ otherwise; so $\bar{g}(u)@Close_{\downarrow}(\{b\}) = \langle\!\langle m_b \rangle\!\rangle$ if u contains a d, and is $\emptyset$ otherwise. Therefore, in all cases,

$$\bar{g}(u@Close_{\downarrow}(\{b\}))@Close_{\downarrow}(\{b\}) = \bar{g}(u)@Close_{\downarrow}(\{b\})$$

– For $t = c$: This case is exactly analogous to that of $t = b$.
– For $t = d$: For all u, $\bar{g}(u)@Close_{\downarrow}(\{d\}) = \emptyset$, so

$$\bar{g}(u@Close_{\downarrow}(\{d\}))@Close_{\downarrow}(\{d\}) = \bar{g}(u)@Close_{\downarrow}(\{d\})$$

The partial order of times, as defined above, is not a lattice. We can, however, give a variant of the example in which it is. We modify the partial order by placing a above b and c (and therefore above d as well); we do not modify the function $\bar{g}$. The argument that Condition 1 fails for the frontier $\{b, c, d\}$ but holds for $Close_{\downarrow}(\{t\})$ when $t \in \{b, c, d\}$ is exactly as above. It remains to check that Condition 1 holds for $Close_{\downarrow}(\{t\})$ when $t = a$.

– For $t = a$: For all u, $\bar{g}(u@Close_{\downarrow}(\{a\})) = \bar{g}(u)$, so

$$\bar{g}(u@Close_{\downarrow}(\{a\}))@Close_{\downarrow}(\{a\}) = \bar{g}(u)@Close_{\downarrow}(\{a\})$$

4.6 Another Perspective on ϕ and Its Properties

Intuitively, we may expect ϕ to have additional properties beyond Condition 1, and such properties are sometimes useful for working with ϕ. For example, we may expect that, for all e, $\phi(e)(T) = T$, since the initial state of a node and its inputs (and their exact interleaving) determine its outputs. We may also expect $\phi(e)$ to be monotonic, since intuitively knowing more of the input cannot remove information about the output. Furthermore, given a function $\phi(e)$ that is not necessarily monotonic, we could define a new monotone function $\phi'(e)$ by

$$\phi'(e)(f) = \cup_{f' \subseteq f} \phi(e)(f')$$

Finally, we may expect that $\phi(e)$ distributes over intersections. This property implies both $\phi(e)(T) = T$ and the monotonicity of $\phi(e)(T)$. We formulate it as follows:

Condition 3. *For all $e \in E$, for any index set X and family of frontiers f_x for $x \in X$, $\phi(e)(\cap_{x \in X} f_x) = \cap_{x \in X} \phi(e)(f_x)$.*

In the remainder of this section, we present another way of looking at frontier transformers. While $\phi(e)$ may be seen as going from inputs to outputs, the alternative perspective is based on reasoning in the opposite direction, from outputs to inputs. We show that the two perspectives yield equivalent results; in our opinion, this equivalence makes frontier transformers (and Condition 3) even more compelling.

Suppose that, for a node p and an outgoing edge e, we are given a function R_0 from times to frontiers, with the property (informally) that knowing p's inputs at $R_0(t)$ suffices for knowing its outputs on e at t. This function induces a monotone function $R(t) = \cup_{t' \leq t} R_o(t')$, with the property that knowing p's inputs at $R(t)$ suffices for knowing its outputs on e up to t, as the following condition asserts.

Condition 4. *If* $g(p)(h) = (\ldots, \langle \ldots e_i \mapsto \mu_i \ldots \rangle)$ *and* $g(p)(h@R(t)) = (\ldots, \langle \ldots e_i \mapsto \mu'_i \ldots \rangle)$ *then* $\mu_i@(Close_\downarrow(\{t\})) = \mu'_i@(Close_\downarrow(\{t\}))$.

Going forward, we prefer to work with R rather than R_0, because we have not set out the notation to work directly with R_0, and because knowing the output only at a time t and not at the times below t may sometimes be useless, in particular in the context of differential computation [10]. The fact that R is (or may be) generated from some function R_0 is reflected in the following monotonicity condition.

Condition 5. *If* $t' \leq t$ *then* $R(t') \subseteq R(t)$.

Every function R induces a function $\phi(e)$, and conversely every function $\phi(e)$ induces a function R, as follows. Let us write $\mathcal{F}$ for the function that maps R to $\phi(e)$ and $\mathcal{G}$ for the function that goes in the opposite direction. For $\rho : T \to F$ and $\psi : F \to F$, we set:

$$\mathcal{F}(\rho)(f) = \{t \mid \rho(t) \subseteq f\}$$

and

$$\mathcal{G}(\psi)(t) = \cap\{f \mid t \in \psi(f)\}$$

We obtain that the conditions on $\phi(e)$ and those on R are exactly equivalent, and that the functions $\mathcal{F}$ and $\mathcal{G}$ are anti-monotone and inverses of each other:

Proposition 9.

- *If* $\phi(e) = \mathcal{F}(R)$ *and* R *satisfies Conditions 4 and 5 then* $\phi(e)$ *satisfies Conditions 1 and 3.*
- *Conversely, if* $R = \mathcal{G}(\phi(e))$ *and* $\phi(e)$ *satisfies Conditions 1 and 3 then* R *satisfies Conditions 4 and 5.*

Proposition 10.

- *If* $\phi(e)(f) \subseteq \phi'(e)(f)$ *for all* f*, then* $\mathcal{G}(\phi'(e))(t) \subseteq \mathcal{G}(\phi(e))(t)$ *for all* t.
- *If* $R(t) \subseteq R'(t)$ *for all* t*, then* $\mathcal{F}(R')(f) \subseteq \mathcal{F}(R)(f)$ *for all* f.

Proposition 11.

- *For all* f, $\phi(e)(f) = \mathcal{F}(\mathcal{G}(\phi(e)))(f)$.
- *For all* t, $R(t) = \mathcal{G}(\mathcal{F}(R))(t)$.

The following example illustrates that Condition 3 is needed in order for us to obtain $\phi(e)(f) = \mathcal{F}(\mathcal{G}(\phi(e)))(f)$, as we do in Proposition 11. Distributivity over finite intersections would not suffice.

Example 2. Suppose that the set of times T consists of the integers (including the negative ones), and that $\phi(e)(f) = T$ if $f \neq \emptyset$ and $\phi(e)(\emptyset) = \emptyset$. Note that $\phi(e)$ distributes over all finite intersections but not over all infinite intersections. We obtain that $\mathcal{G}(\phi(e))(t) = \cap\{f \mid t \in \phi(e)(f)\} = \emptyset$, since $t \in \phi(e)(f)$ for all non-empty f, but the intersection of all non-empty f is empty. Further, we obtain that $\mathcal{F}(\mathcal{G}(\phi(e)))(f) = \{t \mid \mathcal{G}(\phi(e))(t) \subseteq f\} = \{t \mid \emptyset \subseteq f\} = T$, for all f. In sum, $\mathcal{F}(\mathcal{G}(\phi(e)))$ is strictly bigger than $\phi(e)$ in this example.

From a semantics perspective, a frontier is a predicate, and a frontier transformer $\phi(e)$ is a predicate transformer. Curiously, our predicate transformers go from inputs to outputs; generally the opposite is true. Nevertheless, much of the material in this section is part of the general theory of predicate transformers (e.g. [12, p. 83]), not specific to our setting. An exception is the correspondence between Conditions 1 and 4, in Proposition 9.

5 Main Results

In this section we present our main results. We start with an informal discussion of the results which leads to a few definitions, continue with some auxiliary lemmas, then state our main theorem.

Throughout, we assume a function ϕ that satisfies Condition 1. This condition is purely local: it refers to the behavior of each node in isolation. In this section, we use it in order to obtain global guarantees for an entire system.

5.1 Informal Discussion and Definitions

Our main theorem considers the messages that each node p receives within a frontier $D(p)$, possibly a different frontier for each node. Initially, however, let us consider the simple case in which $T = \{\text{“Public”}, \text{“Secret”}\}$, with “Public” $\leq$ “Secret”, and $D(p) = \{\text{“Public”}\}$ for all p. In this case, we can derive that each node's history is independent of any secrets, even if queues may contain secrets initially and even if nodes can generate secrets in response to public messages.

More precisely, suppose that $\sigma = \langle\!\langle s_0, s_1, \ldots \rangle\!\rangle$ is a behavior of the system with initial values for the queues Q_0. Suppose further that HQ_0 is such that $HQ_0(e)@\{\text{“Public”}\} = Q_0(e)@\{\text{“Public”}\}$ for all e, that is, that Q_0 and HQ_0 coincide on public messages. Then there exists an alternative behavior $\hat{\sigma} = \langle\!\langle \hat{s_0}, \hat{s_1}, \ldots \rangle\!\rangle$ with initial values HQ_0 such that, if p has respective histories h and $\hat{h}$ in two corresponding states s_i and $\hat{s_i}$, then $h@\{\text{“Public”}\} = \hat{h}@\{\text{“Public”}\}$. In this alternative behavior, each node has no information about messages outside “Public”, not even that they exist at all.

Recall that, in Section 2, the definition of the action *MessR* says that, given a sequence of messages $u{\cdot}m{\cdot}v$, a node p is allowed to process m when there is no message n ahead of m (so, in u) with $\mathit{time}(n) \leq \mathit{time}(m)$. Although motivated by other applications, this specification of *MessR* seems attractive from an information-flow perspective. It enables a system to produce the same behavior

at $time(m)$ independently of data at higher and unrelated levels. For example, given the queue $n{\cdot}m$ where $time(n)$ = "Secret" and $time(m)$ = "Public", the node p can process m as though n was not there.

Going beyond the special case where D is constant across nodes, we would want that a node p gets no information about messages outside $D(p)$ from messages in $D(p)$. For this purpose, we would assume that Q_0 and HQ_0 coincide on $D(p)$ for edges going into p, and would reason that for every behavior σ with Q_0 there is an alternative behavior $\hat{\sigma}$ with HQ_0 that yields the same histories filtered to $D(p)$ at each node p. Thus, messages at $D(p)$ are fixed, and those outside $D(p)$ differ between σ and $\hat{\sigma}$.

However, not all possible mappings of nodes to frontiers constitute reasonable values for D. For instance, suppose that $D(p) = \{\text{"Public"}\}$, $D(q) = \{\text{"Public"}, \text{"Secret"}\}$, and p has sent some messages to q on a direct edge e from p to q. Any secrets that p has sent to q will be apparent in q's history, and corresponding actions at p must be present in any alternative behavior. Such examples suggest that, when there is an edge from p to q, perhaps we should require that $D(q) \subseteq D(p)$.

Still, this requirement is not quite satisfactory in that it does not consider the dependence of p's outputs on e on p's inputs. Treating this dependence via the function ϕ, we amend the requirement to $D(q) \subseteq \phi(e)(D(p))$. Thus, the frontier at q is included in the frontier determined on e by the frontier at p.

In sum, we arrive at the following definitions:

- We say that a function D from P to F is *coherent* if, whenever $p, q \in P$, $e \in E$, $src(e) = p$, and $dst(e) = q$, $D(q) \subseteq \phi(e)(D(p))$.
- We say that two functions Q_0 and HQ_0 from E to M^* are *equivalent up to* D, and write $Q_0 \simeq HQ_0$, if for all $q \in P$ and $e \in E$ with $q = dst(e)$, $Q_0(e)@D(q) = HQ_0(e)@D(q)$.

We have studied weaker but sound requirements in which we consider not only the static graph topology but also what messages are actually sent. We have also studied the possibility of D being state-dependent, as explained in Section 6. In this paper we do not develop those more sophisticated variants, for simplicity.

5.2 Lemmas

Our first auxiliary lemma relates g, local states, queues, and local histories. It relies on definitions of properties Inv_{LocH} and Inv_{QH}, which it asserts are invariants. Property Inv_{LocH} says that the local state of a node is the local state obtained by applying g to its history. Property Inv_{QH} similarly relates the contents of a queue $Q(e)$ to what is obtained by applying g to the history of e's source. We do not quite have $\Pi_e g(p)(H(p)) = Q(e)$, however, for three reasons:

- the initial value of $Q(e)$ must be added ahead of the result of applying g to the history of e's source, on the left of this equation;
- the messages that e's destination has consumed, which are in its history, must be added ahead of $Q(e)$, on the right;

– finally, reorderings are possible, because of the definition of *MessR*, so we should use a reordering relation rather than an equality.

We arrive at the following definitions and lemma:

– Let Inv_{LocH} be

$$\forall p \in P. \Pi_{\mathrm{Loc}} g(p)(H(p)) = LocState(p)$$

– Let Inv_{QH} be:

$$\forall p, q \in P, e \in E \text{ such that } src(e) = p \wedge dst(e) = q.$$
$$(Q_0(e) \cdot \Pi_e g(p)(H(p))) \hookrightarrow (\langle m \mid (e, m) \in H(q) \rangle \cdot Q(e))$$

– Let Inv_{LocQH} be the conjunction of Inv_{LocH} and Inv_{QH}.

Lemma 1. *ISpec*(Q_0) *implies* $\Box\, Inv_{\mathrm{LocQH}}$.

Our second lemma is motivated by the definition of *HQ* in Section 5.3 below. There, we consider a sequence of messages defined as a subtraction. The lemma implies that the subtraction never resorts to the clause $\emptyset - m = \emptyset$; in other words, the sequence from which we are subtracting contains all the elements of the sequence that we are subtracting, and with at least the same multiplicity.

Lemma 2. *Assume that* $Q_0 \simeq HQ_0$ *and that* D *is coherent. Let* $p = src(e)$ *and* $q = dst(e)$. *Let* $\mu = HQ_0(e) \cdot \Pi_e g(p)(H(p)@D(p))$ *and* $\nu = \langle m \mid (e, m) \in H(q) \rangle$. *Then ISpec*$(Q_0)$ *implies* $\Box(\mu \cdot u - \nu@D(q)) = (\mu - \nu@D(q)) \cdot u$, *for all* u.

5.3 Main Theorem

Our main theorem relies on a way of mapping one state to another state. Specifically, given state functions *LocState*, Q, and H, we define new state functions *HLocState*, *HQ*, and *HH*. We then show that if a behavior satisfies *ISpec*(Q_0) then the behavior induced by the mapping satisfies *ISpec*(HQ_0).

As in other work with TLA (e.g., [8, Section 8.9.4]), we phrase the theorem in terms of formulas and substitutions rather than in terms of behaviors. For any expression *Exp*, we write $\overline{Exp}$ for the result of applying the substitution $[HLocState/LocState, HH/H, HQ/Q]$ to *Exp*.

We let:

$$HLocState(p) = \Pi_{\mathrm{Loc}} g(p)(H(p)@D(p))$$

$$HQ(e) = HQ_0(e) \cdot \Pi_e g(p)(H(p)@D(p)) - \langle m \mid (e, m) \in H(q) \rangle @D(q)$$
$$\text{where } p = src(e), q = dst(e)$$

$$HH(p) = H(p)@D(p)$$

According to these definitions, *HLocState*(p) is obtained by applying $g(p)$, much as in Inv_{LocH}, but filtering the history with $D(p)$. Intuitively, *HLocState*(p) is

intended to be the local state that p would reach if it only saw messages with times in $D(p)$. Similarly $HQ(e)$ aims to describe the contents of $Q(e)$ in an alternative reality in which the source of e would see only messages with times in $D(p)$ and the destination of e would only consume messages in $D(q)$. Its definition has many of the same ingredients as Inv_{QH}. Finally, $HH(p)$ is simply the part of p's local history that is limited to messages with times in $D(p)$.

We obtain:

Theorem 1. *Assume that $Q_0 \simeq HQ_0$ and that D is coherent. Then $ISpec(Q_0)$ implies $\overline{ISpec(HQ_0)}$.*

The following corollary reformulates the theorem in terms of a behavior σ and an alternative behavior $\hat{\sigma}$. It also considers the case where the local history of some node p in σ contains only messages with times in $D(p)$. The corollary states that the node would have exactly the same history in the alternative behavior $\hat{\sigma}$. Thus, the history does not allow p to differentiate σ and $\hat{\sigma}$.

Corollary 1. *Assume that $Q_0 \simeq HQ_0$ and that D is coherent. For every behavior $\sigma = \langle\!\langle s_0, s_1, \ldots \rangle\!\rangle$ that satisfies $ISpec(Q_0)$ there exists a behavior $\hat{\sigma} = \langle\!\langle \hat{s_0}, \hat{s_1}, \ldots \rangle\!\rangle$ that satisfies $ISpec(HQ_0)$ and such that, for all $p \in P$, if $H(p)$ has the value h in s_i then it has the value $h@D(p)$ in $\hat{s_i}$.*

If in addition, for some $p \in P$, σ satisfies $\Box(H(p) = H(p)@D(p))$, then $H(p)$ has the same sequence of values in σ and in $\hat{\sigma}$.

While differences in models make precise comparisons difficult, the properties that these results express resemble non-interference and its possibilistic variants, such as restrictiveness [9, Section 2.2.2]. For instance, restrictiveness talks about adding or deleting "high-level inputs" to a system trace; in our results, the change from Q_0 to HQ_0 can essentially serve that purpose.

5.4 A Small Example

We close this section with an application of Theorem 1 and Corollary 1. It is a trivial exercise, but illustrates how the results can be instantiated.

Consider a simple graph with nodes p_0, p_1, and p_2, with edges e_1 and e_2 from p_0 to p_1 and p_2, respectively, plus an inert node q with an edge e_0 from q to p_0. Initially, $Q(e_0)$ contains messages for two unrelated times t_1 and t_2 that represent private data for two users U_1 and U_2 (as in Section 4.3); $Q(e_1)$ and $Q(e_2)$ are initially empty. Suppose that p_0 demultiplexes the payload of those messages, applies to them a state-independent function, and strips the time information which is not needed at p_1 and p_2. Formally, all of p_0's outputs are in a third, unrelated time *null*.

We still have $\phi(e_1)(\{t_1\}) = T$ and $\phi(e_2)(\{t_2\}) = T$, and we also have $\phi(e_1)(\{t_2\}) = \emptyset$ and $\phi(e_2)(\{t_1\}) = \emptyset$. Since q has no incoming edges, we can take $\phi(e_0)(f) = T$ for all f.

Therefore, we can satisfy the coherence criterion for the function D by letting $D(q) = T$, $D(p_0) = \{t_1\}$, $D(p_1) = T$, and $D(p_2) = \emptyset$. Suppose further that

σ is a behavior of the system with the given initial messages in $Q(e_0)$. Then, according to Corollary 1, there exists another behavior $\hat{\sigma}$ with the same initial messages in $Q(e_0)$ at time t_1 but arbitrary ones at time t_2 (because $D(p_0) = \{t_1\}$). Moreover, $Q(e_1)$ is initially empty in $\hat{\sigma}$ (because $D(p_1) = T$), but the initial contents of $Q(e_2)$ are arbitrary (because $D(p_2) = \emptyset$). It follows from Corollary 1 that the local history at p_1 is identical in σ and $\hat{\sigma}$. In other words, this local history does not allow p_1 to infer anything about which messages at time t_2 are initially present on e_0.

Some alternative choices of D also satisfy the coherence criterion but lead to different results, in particular showing that, symmetrically, p_2 cannot infer anything about which messages at time t_1 are initially present on e_0.

6 Conclusion

In this paper, we study how a dataflow model of computation, timely dataflow, can offer information-flow properties. The required enhancements include the use of functions that express dependencies between inputs and outputs at each node. They are consistent with the possibility that each node operates over a distinct set of virtual times. We leave for further work the enforcement or checking of those dependencies. In the context of Naiad, programming conventions have sometimes been used for ensuring the expected properties of the could-result-in relation; those could probably be extended and codified into information-flow type systems or other static analyses. We also leave for further work the study of declassification and of quantitative information-flow properties, which should be helpful in applications. Although Naiad remains a research artifact, it is already a substantial, efficient system on which non-trivial applications have been developed, but not, to date, with consideration of security and privacy properties. Beyond Naiad, more broadly, there seems to be growing interest in mandatory access control, information-flow control, and their applications in modern data-parallel systems (e.g., [13,6]).

As mentioned in the Introduction, this work stems from a larger effort to understand, improve, and apply timely dataflow. We close this paper with a brief discussion of some of our recent and ongoing work, and how it relates to security.

Section 2 is based on the original description of the timely dataflow model of computation in the context of Naiad [11], and on another paper (in preparation) that studies the model in more generality and detail. In particular, the model includes completion notifications, which tell a node when it will no longer see messages for a given time, and which require a careful definition and analysis of the could-result-in relation. Other features of the model include external input and output channels. We omit these aspects of timely dataflow here, in order to simplify the presentation of this paper, though we have considered their information-flow aspects. Interestingly, completion notifications introduce flows of information "at a distance" (not necessarily from neighbor to neighbor in a dataflow graph), via the run-time system that tracks the progress of the computation and delivers those notifications.

A further paper (also in preparation) explores fault-tolerance in the timely dataflow model. Over the years, connections between non-interference and fault-tolerance have been identified (e.g., [16,15,14]); perhaps it is time to revisit them. Much of the machinery that we present in this paper arose in our work on fault-tolerance, in a more general, more dynamic form. In particular, there, the function D that maps a node to a set of times is state-dependent, rather than static. "Undo computing" [7], which restores system integrity after an intrusion by undoing changes made by an adversary while preserving legitimate user actions, may be an intriguing area of application for this ongoing work.

Acknowledgments. We are grateful to our coauthors on work on Naiad for discussions that led to this paper, and to Gordon Plotkin for pointing out the connection with predicate transformers.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
2. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
3. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
4. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (1985)
5. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)
6. Khan, S.M., Hamlen, K.W., Kantarcioglu, M.: Silver lining: Enforcing secure information flow at the cloud edge. In: 2014 IEEE International Conference on Cloud Engineering, pp. 37–46 (2014)
7. Kim, T., Wang, X., Zeldovich, N., Kaashoek, M.F.: Intrusion recovery using selective re-execution. In: 9th USENIX Symposium on Operating Systems Design and Implementation, pp. 89–104 (2010)
8. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
9. McLean, J.: Security models. In: Marciniak, J. (ed.) Encyclopedia of Software Engineering. Wiley & Sons (1994)
10. McSherry, F., Murray, D.G., Isaacs, R., Isard, M.: Differential dataflow. In: CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research (2013)
11. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: A timely dataflow system. In: ACM SIGOPS 24th Symposium on Operating Systems Principles, pp. 439–455 (2013)
12. Plotkin, G.: Domains, the so-called Pisa notes (1983), `http://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps`.
13. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: Security and privacy for MapReduce. In: Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, pp. 297–312 (2010)

14. Rushby, J.: Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center (June 1999)
15. Simpson, A., Woodcock, J., Davies, J.: Safety through security. In: Proceedings of the 9th International Workshop on Software Specification and Design, pp. 18–24. IEEE Computer Society (1998)
16. Weber, D.G.: Formal specification of fault-tolerance and its relation to computer security. In: Proceedings of the 5th International Workshop on Software Specification and Design, pp. 273–277. ACM (1989)
17. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, pp. 15–28 (2012)