2018 IEEE 31st Computer Security Foundations Symposium

Assuming You Know: Epistemic Semantics of Relational Annotations for Expressive Flow Policies

Andrey Chudnov
D. E. Shaw & Co., L.P.
New York NY, USA

David A. Naumann
Stevens Institute of Technology
Hoboken NJ, USA

***Abstract*—Many high-level security requirements are about the allowed flow of information in programs, but are difficult to make precise because they involve selective downgrading. Quite a few mutually incompatible and ad-hoc approaches have been proposed for specifying and enforcing downgrading policies. Prior surveys of these approaches have not provided a unifying technical framework. Notions from epistemic logic have emerged as a good approach to policy semantics but are considerably removed from well developed static and dynamic enforcement techniques. We develop a unified framework for expressing, giving meaning and enforcing information downgrading policies that builds on commonly known and widely deployed concepts and techniques, especially static and dynamic assertion checking. These concepts should make information flow accessible and enable developers without special training to specify precise policies. The unified framework allows to directly compare different policy specification styles and enforce them by leveraging existing techniques.**

I. Introduction

A longstanding challenge in computer security is how to specify and enforce confidentiality requirements. Confidentiality is difficult to define precisely, owing to the subtlety of information flow in systems and the subtlety of requirements which usually involve downgrading (i.e., flows of partial information under designated conditions). An example of such requirements is encrypted media distribution where a user does not receive the decryption key until they have paid for the movie, and they only learn the keys for the movies for which they paid [3]. Those subtleties also make it difficult to achieve high assurance of enforcement at reasonable cost, meaning both development cost and runtime performance. The story is similiar for integrity requirements, and the two are intertwined [5], [24], but in this paper we streamline the development by considering only confidentiality.

The notion of knowledge has emerged as a good approach to specification that enables mathematically robust semantics for confidentiality with downgrading (e.g. [11], [5], [9], [22]). There is a mature theory of epistemic logic, including algorithms for model checking that may be suitable for system models. It is less clear how to apply such logics effectively at the level of software interfaces and implementations.

Motivated by practical considerations, a number of works propose specifications by means of labels on variables and other program elements, together with special features such as declassification statements. Studies using epistemic logic to specify confidentiality have used relatively simple logical formulas [9] so it may be that a less general logical formalism, retaining epistemic semantics, can suffice for software specifications. Assertions and pre-post contracts are familiar to developers of systems concerned with high assurance, and researchers are exploring generalizations like relational Hoare logic which has already been shown to provide a flexible and expressive means to specify downgrading policies in the context of "batch programs" (i.e., pre-post observations) [38].

This paper follows up on the proposal by Chudnov et al. [25] to use intermediate annotations—relational assumptions and assertions—both to specify and to enforce security. In Section II we review their policy specification approach through examples. Unfortunately, the semantics used in [25] is intricate and significantly different from security definitions used in prior work. They justify the semantics by (a) showing it coincides with standard termination-insensitive noninterference, for simple pre-post policies, and (b) conjecturing that some knowledge based semantics of declassification (specifically, [11], [7]) can be captured in their semantics using suitable annotations.

Our *first contribution* is a novel epistemic semantics for this specification approach (Section IV). We do not, however, make a direct connection between the cited works, because our development is in the setting of reactive programs—which caters to requirements for confidentiality of information, not all of which is available in the program's initial state or input (Section III). Unlike those works, we explicitly model inputs and outputs, which are the observables. (As usual, the adversary is omniscient about the program and its semantics.)

Our *second contribution* is to define a notion of safety, conceptually similar to the annotation-oriented semantics of [25] but adapted to the reactive model (Section V), and to prove that the safety condition implies the epistemic security condition (Section VI).

Our *third contribution* is to sketch how our safety semantics can be enforced not only by monitoring (as done in [25]) but also by static verification in a relational Hoare logic (Section VII). We wrap up with more discussion of related work (Section VIII) and conclude (Section IX). Some proofs are in the appendix.

II. Declassification by example

We introduce our specifications, and illustrate their expressive power, with a few examples, written in a small imperative

DOI 10.1109/CSF.2018.00021

reactive language inspired by [18]. We add relational assertions and assumptions, inspired by [25], for specifying information flow requirements. The language is formally defined in section III.

Our first example is a randomized response sampling algorithm (Figure 1). The purpose of the algorithm is to provide differential privacy and plausible deniability to the respondents to a sensitive yes/no question, preserving the statistical characteristics of the underlying data [30]. The core idea is that each respondent provides either an honest or a random answer depending on a random coin flip. Our example samples the raw responses and discloses them according to the aforementioned approach. Here we use three communication channels, Private (P) input that feeds binary private responses to the sampler, Random (R) input providing a source of secure random bits, and Public (L) output, that expects the anonymized responses. Programs consist of input handlers, one for each input channel. Handlers run to completion and optionally produce output. The first handler (input^P) is the input sampler. The second handler ensures the first always has two fresh random bits. For simplicity, we source randomness from one channel, and thus have to ensure there are two fresh random bits, stored in variables r_1 (for the coin flip) and r_2 for the randomized reply.

For simplicity, this contrived program is intended to be used in an environment where two bits of randomness are sent prior to each response bit. Loss of additional random bits would not matter but dropping response bits would be unsatisfactory.

We can ensure security of this program by enforcing the following information flow policy. Private data is only disclosed to the public *under the condition* that r_1 is true. We can formalize this policy by labeling channels and arranging the labels in a partial order, such that for any labels ℓ_1, ℓ_2 related by the ordering $\ell_1 \preceq \ell_2$ information from the input channel with a label ℓ_1 can flow to the output channel with a label ℓ_2. In the example, the labels are ordered such that $L \preceq P$ and $L \preceq R$. This baseline policy prevents direct disclosure of the both the randomness and the responses to the public observer. However, it also prevents the legitimate disclosure as well. We resolve this issue by relaxing the policy in two places in the program using downgrading *annotations* highlighted in Figure 1. The annotation $\mathsf{assume}^L\, \mathbb{A}r$ declassifies the received bit r to the public label L. The notation alludes to the standard two-run semantics of dependency: the formula "$\mathbb{A}r$" is read "agreement on r", and the policy says if two runs agree on r then outputs on L also agree.

Note that this is different from saying that the random channel is public, as that would make the whole random stream visible to the attacker. The annotation $\mathsf{assume}^L \mathbb{B}r_1 \Rightarrow \mathbb{A}p$ relabels the private response bit p with the label L, but only in the case the value of r_1 in that program execution is true.

In this illustrative language, program variables are all initialized to 0 at system start, and a handler parameter gets initialized with each input.

Whether this is a reasonable program and policy depends on the statistical properties of the channel R. In general,

```
input^P p.  if ready > 1 then
              assume^L B r1 ⇒ A p;
              if r1 then o := p; else o := r2; fi
              output^L o; ready := 0; fi
input^R r.  assume^L A r;
            if ready = 0 then r1 := r;
            else if ready = 1 then r2 := r; fi
            ready := ready + 1;
```

Fig. 1. Randomized response sampling algorithm

the policy specification needs to be justified in terms of the quantitative notions of differential privacy. However, that also requires reasoning about the environment in which the program is run and the statistical properties of its inputs. In contrast, the specification in the code expresses what the observer is allowed to learn: the R values and the P values, but only under a certain condition. This can be understood in non-quantitative, epistemic terms as we formalize later. Together with the assumption of randomness of the R inputs, this gives us the security guarantee. Note that this is not an isolated example of this issue: the security of the well-known in the literature password checking example, where only the outcome of the password test but not the password itself can be declassified, depends in a large part on the length of passwords and their distribution. A typical policy can be expressed, in our notation, as $\mathbb{A}numguesses \wedge (\mathbb{B}(numguesses < 10) \Rightarrow \mathbb{A}(guess = pass))$. If the set of possible secret values is small enough, even disclosing the result of this small number of tries is unacceptable as this allows the attacker to simply explore the space exhaustively.

These examples express downgrading in terms of current values of program variables, although only inputs and outputs are considered to be observable. In the next example, the policy refers directly to recent inputs. Instead of the agreement form $\mathbb{A}r$ for a variable, it uses the agreement form $\mathbb{A}\ell@n$ that expresses agreement on the nth value on input channel ℓ.

The next example is a simple packet sniffer that logs the source IP addresses of packets while ensuring confidentiality of payloads (Figure 2). Packets are read off the secret channel S byte-by-byte; IP addresses are logged on the public channel P. The intended policy is that only the address, stored in the 12-th to 15-th bytes of the packet, is logged. Disclosing any other part of the packet is against the policy.

Since the handler processes packets byte-by-byte, we also need to perform some bookkeeping to track where in the packet we currently are and refer accurately to the position of the IP address in the packet. This is done by updating two auxiliary variables: b counts the total number of bytes received so far and p the number of bytes received in the current packet. In order to accurately specify this policy we also need a way to refer to the elements of the input stream. This is done using a novel annotation type $\mathsf{assume}^L\, \mathbb{A}S@n$ which downgrades the n-th (0-based) input on channel S to label L. Using this new annotation together with logical

```
input^S x.  assume^P 𝔸S@b − p + 12 ∧ ... ∧ 𝔸S@b − p + 15;
            if p > 11 ∧ p < 16 then output^P x;
            if p = 2 then l := x ≪ 8;
            if p = 3 then l := l + x;
            output^S x;
            p := p + 1;
            b := b + 1;
            if p > l then p := 0;
```

Fig. 2. A simple packet sniffer

conjunctions and the ability to refer to the values of program variables, allows us to write the downgrading annotation $\mathsf{assume}^P\ \mathbb{A}S@b-p+12 \wedge \ldots \wedge \mathbb{A}S@b-p+15$ which says that every byte in the stream with offset between $b-p+12$ and $b-p+15$[1] should be declassified on the public channel.

III. Background: events, programs, annotations

As illustrated in the preceding examples, we formulate policy in terms of a fixed labeling of channels and a fixed flow relation on labels, together with annotations to express conditional flows that would not be allowed by the baseline policy. This section begins the technical development by formalizing these building blocks.

We assume given a partially ordered set $\mathcal{L}$ of security levels, and write $\ell \preceq \ell'$ for the ordering. Given the availability of declassifying assumptions, one interesting instantiation is to use a discretely ordered set.

Events t are given by the grammar

$$t ::= \mathsf{in}^\ell n \mid \mathsf{out}^\ell n \mid \bullet \qquad \text{where } \ell \in \mathcal{L},\ n \in \mathbb{N}$$

Identifiers t, u, v range over events. The event $\mathsf{in}^\ell n$ represents the input of value n on ℓ's input channel; $\mathsf{out}^\ell n$ is for ℓ's output channel. The ***tick*** event $\bullet$ is neither output nor input, it just lets us associate an event with every transition, to streamline notation.

An event t is ***visible*** at ℓ if t is an input or output at some level $\ell' \preceq \ell$. Let $\mathsf{In}^\ell = \{\mathsf{in}^{\ell'} n \mid \ell' \preceq \ell \text{ and } n \in \mathbb{N}\}$, Let $\mathsf{Out}^\ell = \{\mathsf{out}^{\ell'} n \mid \ell' \preceq \ell \text{ and } n \in \mathbb{N}\}$ and $\mathsf{Ev}^\ell = \mathsf{In}^\ell \cup \mathsf{Out}^\ell$. For a sequence ts of events we write $\mathsf{vis}^\ell(ts)$ for the subsequence of ℓ-visible ones, i.e., those in Ev^ℓ.

A. Programs

A program consists of a set of input handlers, one for each security level. The program consumes a sequence of inputs, handling each one in turn. The syntax $\mathsf{input}^\ell x.\, c$ is for handler associated with channel ℓ, with body c, in which variable x is initialized to the input value. A handler may produce zero or more outputs, which may be on different channels from its input. If a handler diverges, no further input is consumed. Programs are ***input total*** in the sense that unless the last handler is diverging, the next input event can be consumed.

Annotated programs

e	$::= n \mid x \mid e \oplus e \mid \neg e$	expression
	where $n \in \mathbb{Z}, x \in \mathit{Vars}, \oplus \in \{+, *, =, \leq, \wedge, \ldots\}$	
c	$::= \mathsf{skip} \mid x := e \mid \mathsf{output}^\ell\ e$	command
	$\mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid c\,;c$	
	$\mid \mathsf{assert}^\ell\ \Phi \mid \mathsf{assume}^\ell\ \Phi$	annotation
H	$::= \mathsf{input}^\ell x.\, c$	handler

An ℓ***-annotation*** is a command of the form $\mathsf{assert}^{\ell'}\ \Phi$ or $\mathsf{assume}^{\ell'}\ \Phi$ such that $\ell' \preceq \ell$. Annotations have no influence on the runtime behavior of the program. Strictly speaking, each occurrence of an annotation command needs to have a unique tag, but we omit that for readability.[2] The ℓ-annotations that are assumptions specify what the observer at level ℓ—seeing inputs and outputs on channels $\ell' \preceq \ell$—is allowed to learn. Assertions are only needed for enforcement (Section V). As shown by the introductory examples, the handler for some level ℓ may have annotations for unrelated levels.

Relational formulas Φ for annotations

$\varphi ::= e \mid \varphi \vee \varphi \mid \neg\varphi \mid \forall x.\varphi$	state predicate
$\Psi ::= \mathbb{A}e \mid \mathbb{A}\ell@e$	agreement
$\Phi ::= \Psi \mid \mathbb{B}\varphi \mid \mathbb{B}\varphi \Rightarrow \Psi$	basic relational formula
$\Phi ::= \Phi \wedge \Phi$	conjunction of basic formulas

As formalized later, $\mathbb{B}\varphi$ is satisfied in a pair of states if *both* satisfy φ, and $\mathbb{A}e$ is satisfied in a pair of states if they *agree* on the value of e.

B. Program semantics

In the rest of the paper, we consider a fixed program i.e. set of handlers, which is left as an implicit parameter in order to streamline notation.

A ***state*** σ maps variables to natural numbers. At the beginning of execution, all variables are 0. At the start of a handler execution, the designated variable is set to the input value and all other variables are unchanged. ***Configurations*** are of two kinds. A ***receptive*** configuration $\langle\sigma\rangle$ contains just a state σ. An ***active*** configuration $\langle c,\ \sigma\rangle$ represents the execution of a handler, where c represents the current control and σ is a state. The initial configuration is $\langle\sigma_0\rangle$ where σ_0 maps all variables to 0. We write $g \xrightarrow{t} h$ to indicate that g transitions to h with associated event t.

[1] $p - n$ tells us how many more bytes we need to input to see the n-th byte from the beginning if the packet; $b - n$ allows us to refer to the byte the handler will see after the next n inputs. Hence, $b - p + n$ tells us how many more inputs need to be performed to see an n-th byte from beginning of the packet.

[2] Readers of [25] please note: in that work, such tags are provided, using the symbol ℓ which is used here for an entirely different purpose. Also, their term "trace" refers to what we call pre-run.

Transition semantics $g \xrightarrow{t} h$

$$\frac{\mathsf{input}^{\ell} x.c \text{ is the handler for } \ell \qquad \sigma' = (\sigma \,|\, x : n)}{\langle \sigma \rangle \xrightarrow{\mathsf{in}^{\ell} n} \langle c,\, \sigma' \rangle}$$

$$\langle \mathsf{skip},\, \sigma \rangle \xrightarrow{\bullet} \langle \sigma \rangle \qquad \frac{\sigma(e) = n}{\langle \mathsf{output}^{\ell}\ e,\, \sigma \rangle \xrightarrow{\mathsf{out}^{\ell} n} \langle \mathsf{skip},\, \sigma \rangle}$$

$$\frac{\sigma' = (\sigma \,|\, x : \sigma(e))}{\langle x := e,\, \sigma \rangle \xrightarrow{\bullet} \langle \mathsf{skip},\, \sigma' \rangle} \qquad \frac{\sigma(e) \neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ d,\, \sigma \rangle \xrightarrow{\bullet} \langle c,\, \sigma \rangle}$$

$$\langle \mathsf{assume}^{\ell}\ \Phi,\, \sigma \rangle \xrightarrow{\bullet} \langle \mathsf{skip},\, \sigma \rangle \qquad \langle \mathsf{assert}^{\ell}\ \Phi,\, \sigma \rangle \xrightarrow{\bullet} \langle \mathsf{skip},\, \sigma \rangle$$

$$\langle \mathsf{skip}; c,\, \sigma \rangle \xrightarrow{\bullet} \langle c,\, \sigma \rangle \qquad \frac{\langle c,\, \sigma \rangle \xrightarrow{t} \langle c',\, \sigma' \rangle}{\langle c; d,\, \sigma \rangle \xrightarrow{t} \langle c'; d,\, \sigma' \rangle}$$

The first rule consumes an input, transitioning from a receptive configuration to an active one. The updated state $(\sigma \,|\, x : n)$ maps x to the input value n and leaves the other variables unchanged. The other rules all start from an active configuration, and produce either an output event or $\bullet$. The second rule transitions to a receptive configuration from an active one in which the handler has terminated. The remaining rules involve only active configurations.

The output command is the only one that produces an output event. In its rule we write $\sigma(e)$ for the value of expression e in state σ. Comparisons ($=, \leq$) and logical operators ($\wedge, \neg$) return 1 or 0, and non-zero values are interpreted as true. Omitted rules for loops and conditional are standard, as are the last two rules which are for sequencing.

If c is different from skip then we can define $\mathsf{redex}(\langle c,\, \sigma \rangle)$ to be the unique command b such that b is not a sequential composition and either $c \equiv b\,; d$ for some d, or $b \equiv c$. For example, in the penultimate rule the redex is skip. The last rule allows the redex in a sequence to take a step.

The semantics is deterministic and input total.

Lemma 1. For every active configuration there is a unique output or $\bullet$ event and unique successor configuration. For every receptive configuration and every input event there is a unique successor configuration.

C. Program pre-runs and event traces

On the basis of transitions, Bohannon el al [18] define transducers which consume a stream of inputs and produce a stream of outputs. For our purposes it is more convenient to start with an interpretation of programs as admitting a single event trace containing both inputs and outputs. Knowledge is defined in terms of the trace of events a program engages in, but specifications are expressed as program annotations, so we need to work with both computations and event traces.

A ***pre-run*** is a non-empty list of consecutive program configurations, starting with the initial configuration $\langle \sigma_0 \rangle$. In the literature on epistemic logic [9] a "run" is a complete program execution; so we use the term pre-run as we are considering finite prefixes only.

Some notations for lists

We use letters g, h for configurations, and plural identifiers gs, hs for finite sequences thereof—"lists" for short. Likewise we use ts for lists of events, and sometimes ins for lists of input events. We write $|gs|$ for the length of gs, and $\cdot$ for catenation of lists—and also for appending an element at the start or end of a list. We also treat lists as functions from an initial segment of the naturals. So gs_0 is the first configuration and $dom(gs)$ is the set of indices $0, \ldots, |gs| - 1$. We write $\mathsf{last}(gs)$ for $gs_{|gs|-1}$, and $gs \lfloor i$ for the first i elements of gs (i.e., the prefix up to but not including the element gs_i). We write $gs' \geq gs$ to say gs is a prefix of gs'.

For any event list ts, let $\mathsf{inp}(ts)$ to be the subsequence of input events. Every list ins of input events gives rise to a unique pre-run which consumes the inputs or diverges trying to. Before formalizing this, we define the unique sequence of input and output events, called a ***trace*** associated with a pre-run. To avoid confusion with the terms "produce" and "consume", we use a neutral term, admit.

Event trace admitted by a pre-run $gs \Downarrow ts$

$$[\langle \sigma_0 \rangle] \Downarrow [\,] \qquad \frac{gs \Downarrow ts \qquad \mathsf{last}(gs) \xrightarrow{t} g}{gs \cdot g \Downarrow ts \cdot t}$$

A pre-run is just a configuration list that admits a trace.

The pre-runs of the program under consideration comprise the set $\{gs \mid \exists ts.\ gs \Downarrow ts\}$. It is closed under nonempty prefixes but does not contain the empty list $[\,]$. In subsequent definitions and results, quantifications over pre-runs implicitly range over this set.

In light of determinacy, one may expect that a pre-run gs determines a unique ts with $gs \Downarrow ts$. Indeed, in the configuration following an input transition one can see from the handler body which channel's handler is running—unless two channels have identical handlers! Starting in Section V we impose a mild condition that makes handler bodies distinct, but until then we avoid relying on pre-runs determining unique traces.

As an example consider the program with handlers

$$\mathsf{input}^{P} x.\ \mathsf{output}^{Q} x + 1$$
$$\mathsf{input}^{Q} x.\ \mathsf{output}^{Q} y; y := x + y$$

It has a pre-run which admits the trace $ts = [\mathsf{in}^{Q} 3, \mathsf{out}^{Q} 0, \bullet, \bullet, \bullet, \mathsf{in}^{P} 1, \mathsf{out}^{Q} 2, \bullet]$. This trace has inputs $\mathsf{inp}(ts) = [\mathsf{in}^{Q} 3, \mathsf{in}^{P} 1]$. The pre-run has distinct states $(x{:}0, y{:}0)$, $(x{:}3, y{:}0)$ (Q handler), $(x{:}3, y{:}3)$ (Q handler), $(x{:}1, y{:}3)$ (P handler), omitting stutters.

Owing to the possibility of divergence, input totality does not imply that every input sequence can be fully consumed.

However, we can talk about the pre-runs that arise in consuming an input list ins. The gist is that we build gs by executing the handlers on $ins_0, ins_1, \ldots$, until either a handler diverges or all inputs have been consumed. In order to describe gs in a way that is unique in the divergence case, without recourse to infinite sequences, the following result truncates a divergent pre-run at the start of its handler. If some input, say ins_k, causes its handler to diverge, that is described in terms of a series of finite prefixes of the diverging sequence, each beginning with the handler for ins_k, and never reaching a receptive configuration.

Lemma 2. [pre-run from input] For any list ins of input events, there is a unique pre-run gs such that (a) $\mathsf{inp}(gs) \leq ins$, (b) $\mathsf{last}(gs)$ is receptive, and (c) if $\mathsf{inp}(gs) \neq ins$ then there is an infinite sequence G of nonempty lists of active configurations such that, for all i, j we have: $gs \cdot G_i$ is a pre-run, with $\mathsf{inp}(gs \cdot G_i) \leq ins$, and $i < j \Rightarrow G_i < G_j$.

If the handlers for ins all terminate, then gs has all and only the handler executions. In the case that the handler diverges on ins_k, the pre-run gs given by the Lemma has $\mathsf{inp}(gs) = ins\lfloor k$. In that case, notice that $\mathsf{inp}(gs \cdot G_i) = ins\lfloor(k+1)$ for all i.

IV. Knowledge semantics for declassification

In this section we define what it is for a pre-run to be secure. Section IV-A defines the notion of knowledge at a level ℓ: what can be learned by observing channels visible to ℓ. Section IV-B defines the notion of release policy at ℓ, expressed by ℓ-assumptions in the program, and defines security, which says that knowledge is gained only in accord with release policy.

A. Knowledge

Conceptually, the ℓ-observer decides what inputs to provide at levels $\ell' \preceq \ell$, and learns from observing outputs on output channels at levels $\ell' \preceq \ell$. Following Beringer's [16] terminology, the ***major*** pre-run or trace is the actual execution and the ***minor*** ones are the possible executions about which the observer reasons. For a given pre-run gs, the ℓ-observer's knowledge of the inputs (on all channels) is based on their observation of the ℓ-visible inputs and outputs.

Knowledge from a trace k^ℓ

$$\mathsf{k}^\ell(ts) = \{\mathsf{inp}(us) \mid \exists gs.\ gs \Downarrow us \wedge \mathsf{vis}^\ell(ts) = \mathsf{vis}^\ell(us)\}$$

The condition captures knowledge as follows: having observed $\mathsf{vis}^\ell(ts)$, the observer knows that the possible complete inputs ins are those that arise from some pre-run gs of the program, with trace us that is the same to the observer.

The inputs known to be possible are closed under suffixes that are not visible but arise from possible executions. This observation can be made precise as follows. Suppose $fs \Downarrow us$ and $fs \cdot gs \Downarrow us \cdot vs$, with $\mathsf{vis}^\ell(ts) = \mathsf{vis}^\ell(us)$ and $\mathsf{vis}^\ell(vs) = [\,]$. Then $\mathsf{k}^\ell(ts)$ contains not only $\mathsf{inp}(us)$ but also $\mathsf{inp}(us \cdot vs)$.

For the previous example, assuming levels P, Q are incomparable, the Q observer sees $[\mathsf{in}^Q 3, \mathsf{out}^Q 0, \mathsf{out}^Q 2]$ and reasons that the second output does not correspond to any of Q's inputs and, hence, must be derived from P's input of 1.

For another example, consider the program

$$\begin{array}{l} \mathsf{input}^P x.\ y := x;\ \mathsf{output}^P y \\ \mathsf{input}^Q x.\ \mathsf{output}^Q y; \end{array} \tag{1}$$

For the trace $[\mathsf{in}^P 1, \mathsf{out}^P 1, \mathsf{in}^Q 0, \mathsf{out}^Q 1]$, and knowing that all variables are initially 0 and that handlers run to completion before further input is consumed, the Q observer seeing $[\mathsf{in}^Q 0, \mathsf{out}^Q 1]$ can infer that the most recent input on P before her input on Q was the value 1. But she cannot rule out further inputs on P, which would not have Q-visible effect. So the knowledge includes not only the actual inputs $[\mathsf{in}^P 1, \mathsf{in}^Q 0]$ but also inputs like $[\mathsf{in}^P 3, \mathsf{in}^P 1, \mathsf{in}^Q 0, \mathsf{in}^P 4]$. On the other hand, suppose P's output is visible:

$$\begin{array}{l} \mathsf{input}^P x.\ y := x;\ \mathsf{output}^Q 2 \\ \mathsf{input}^Q x.\ \mathsf{output}^P y; \end{array}$$

For the trace in $[\mathsf{in}^P 1, \mathsf{out}^Q 2]$ the Q observer can infer there was a P-input but not its value, and cannot rule out the possibility of $[\mathsf{in}^P 3]$ but can rule out, for example, this one $[\mathsf{in}^P 1, \mathsf{in}^P 1]$ because the handler for the first $\mathsf{in}^P 1$ would have produced Q-visible output.

Lemma 3. (a) If t is not ℓ-visible then $\mathsf{k}^\ell(ts \cdot t) = \mathsf{k}^\ell(ts)$. (b) If t is in Out^ℓ then $\mathsf{k}^\ell(ts \cdot t) \subseteq \mathsf{k}^\ell(ts)$.

Proof. (a) is direct from the definition. For (b), suppose $hs \Downarrow us$ and $\mathsf{vis}^\ell(ts \cdot t) = \mathsf{vis}^\ell(us)$, so that $\mathsf{inp}(us) \in \mathsf{k}^\ell(ts \cdot t)$. To show $\mathsf{inp}(us) \in \mathsf{k}^\ell(ts)$, factor hs as $hs = hs' \cdot g \cdot hs''$ where the transition to g produces t. So we can factor $us = us' \cdot t \cdot us''$ with $hs' \Downarrow us'$ and $hs' \cdot g \Downarrow us' \cdot t$.[3] We have $\mathsf{vis}^\ell(us') = \mathsf{vis}^\ell(ts)$ so $\mathsf{inp}(us') \in \mathsf{k}^\ell(ts)$. Hence, since t is an output, $\mathsf{inp}(us' \cdot t) \in \mathsf{k}^\ell(ts)$. None of us'' are ℓ-visible (owing to $\mathsf{vis}^\ell(ts \cdot t) = \mathsf{vis}^\ell(us)$), so by the observation about suffix closure we have $\mathsf{inp}(us' \cdot t \cdot us'') = \mathsf{inp}(us) \in \mathsf{k}^\ell(ts)$. □

Note that in case t is an ℓ-visible input we should not expect to have $\mathsf{k}^\ell(ts \cdot t) \subseteq \mathsf{k}^\ell(ts)$. The observer is reasoning about what has happened, not what is going to happen. One might formulate a notion similar to $\mathsf{k}^\ell(ts)$ but including also the possible future inputs that ℓ could provide, but this lacks motivation.

The difference $\mathsf{k}^\ell(t) \setminus \mathsf{k}^\ell(ts \cdot t)$ represents what is learned by observing output event t. The baseline policy expressed by labels—noninterference—can be expressed by saying nothing is ever learned: for all ts and all output events t, $\mathsf{k}^\ell(t) \setminus \mathsf{k}^\ell(ts \cdot t) = \emptyset$ or equivalently (in light of Lemma 3), $\mathsf{k}^\ell(t) \subseteq \mathsf{k}^\ell(ts \cdot t)$.

The notion of knowledge defined above is termination or progress-sensitive: $\mathsf{k}^\ell(ts)$ excludes input lists that drive the program to divergence before the last input is handled (recall Lemma 2). In reality, one cannot observe the absence of progress—only stronger properties such as the passage of a fixed amount of time. Moreover, enforcement of progress-sensitive security can be costly and restrictive. So we aim

[3] One may think that in general $hs' \cdot g \Downarrow us' \cdot t$ implies $hs' \Downarrow us'$ but that is only if hs' is non-empty.

for a progress insensitive security property. A simple way to formulate such a property is to entirely disregard diverging pre-runs (as in [3]), but this fails to address the security of visible events prior to divergence. A more sophisticated approach (e.g., [7]) formulates security in a way that essentially declassifies, at each step of execution, the fact whether the program will diverge without producing or enabling the consumption of further visible events. In this approach one formulates a notion of ***progress knowledge*** $\mathsf{k}^\ell_\rightarrow$ defined like k^ℓ except that in addition the observer learns there will be another visible event.

Progress knowledge $\mathsf{k}^\ell_\rightarrow$

$$\mathsf{k}^\ell_\rightarrow(ts) = \{\mathsf{inp}(us) \mid \exists hs, t.\ t \in \mathsf{Ev}^\ell \wedge hs \Downarrow us \cdot t \wedge \mathsf{vis}^\ell(ts) = \mathsf{vis}^\ell(us)\ \}$$

Lemma 4. For any ts, (a) $\mathsf{k}^\ell_\rightarrow(ts) \subseteq \mathsf{k}^\ell(ts)$, and (b) if t is in Out^ℓ then $\mathsf{k}^\ell(ts \cdot t) \subseteq \mathsf{k}^\ell_\rightarrow(ts)$.

Proof. (a) is direct from the definitions. For (b), suppose $ins \in \mathsf{k}^\ell(ts \cdot t)$. Thus there are hs, us with $hs \Downarrow us$, $\mathsf{vis}^\ell(ts \cdot t) = \mathsf{vis}^\ell(us)$, and $ins = \mathsf{inp}(us)$. To show $ins \in \mathsf{k}^\ell_\rightarrow(ts)$, factor $hs = hs' \cdot g \cdot hs''$ and $us = us' \cdot t \cdot us''$ so that $hs' \Downarrow us'$ and $hs' \cdot g \Downarrow us' \cdot t$, and moreover g gives the last occurrence of t in us, so the elements of us'' are not in Ev^ℓ. So we have $\mathsf{vis}^\ell(us') = \mathsf{vis}^\ell(ts)$ and $\mathsf{inp}(us') \in \mathsf{k}^\ell_\rightarrow(ts)$. Since t is an output, $\mathsf{inp}(us' \cdot t) \in \mathsf{k}^\ell_\rightarrow(ts)$ as well. By closure under non-visible suffixes, $\mathsf{inp}(us' \cdot t \cdot us'') = ins \in \mathsf{k}^\ell_\rightarrow(ts)$. □

Progress insensitive security conditions in the literature are roughly equivalent to $\mathsf{k}^\ell(ts \cdot t) \supseteq \mathsf{k}^\ell_\rightarrow(ts)$ holding for all ts and all visible outputs t. We cannot do an exact comparision with, say, ID security of Bohannon et al. [18] because our model is not exactly the same, e.g., they consider $\bullet$ be observable at $\top$. And other works use different program models.

Lemmas 3 and 4 do not address the question of how knowledge is affected by visible input. Intuitively, the ℓ-observer learns nothing from a step in which an ℓ-input is provided. This is evident in the effect on knowledge sets: The believed-possible inputs following a visible input steps are those believed possible before, followed by the visible input, possibly followed by unknown invisible ones.

Lemma 5. If t is an ℓ-visible input then $\mathsf{k}^\ell(ts \cdot t)$ is the set of input lists $ins \cdot t \cdot ins'$ such that $ins \in \mathsf{k}^\ell(ts)$, $\mathsf{vis}^\ell(ins') = [\,]$, and $ins \cdot t \cdot ins'$ is admitted by some pre-run of the program.

B. Release policy and security

Release policy is designated by assumptions in the program. We choose to allow assumptions to refer both to inputs and to the current state. By contrast, knowledge as formulated here is about inputs only. Some policies are naturally expressed in terms of the current state rather than all the inputs that led to that state; it helps avoid the need for the policy specification to duplicate operations also defined in the code. Moreover, for enforcement it is helpful to use assumptions and assertions that refer to the current state. On the other hand, policies that refer only to inputs are less tied to the specific program. The apparent mismatch between policy assumptions and knowledge is resolved by existentially quantifying over the state of the minor execution. In effect, a policy assumption says the observer may not only learn something about the inputs but also learn that the major execution passed through some state that satisfied the assumption.[4]

A relational formula Φ is interpreted as a relation on states paired with input lists. To define that $\mathbb{A}\ell@n$ means agreement on the nth element of the channel-ℓ inputs, we write $\mathsf{ch}^\ell(ts)$ to keep from ts just the events in event list ts at *exactly* level ℓ. By contrast, $\mathsf{vis}^\ell(ts)$ includes events for all $\ell' \preceq \ell$.

Semantics of relational formulas $\sigma, ins \mid \sigma', ins' \models \Phi$

$\sigma, ins|\sigma', ins' \models \mathbb{A}e$ iff $\sigma(e) = \sigma'(e)$

$\sigma, ins|\sigma', ins' \models \mathbb{A}\ell@e$ iff
$|\mathsf{vis}^\ell(ins)| > n$ and $|\mathsf{vis}^\ell(ins')| > n$ imply
$(\mathsf{ch}^\ell(ins))_n = (\mathsf{ch}^\ell(ins'))_n$ where $n = \sigma(e)$

$\sigma, ins|\sigma', ins' \models \mathbb{B}\varphi$ iff $\sigma, ins \models \varphi$ and $\sigma', ins' \models \varphi$

$\sigma, ins|\sigma', ins' \models \mathbb{B}\varphi \Rightarrow \Psi$ iff
$\sigma, ins|\sigma', ins' \models \mathbb{B}\varphi$ implies $\sigma, ins|\sigma', ins' \models \Psi$

$\sigma, ins|\sigma', ins' \models \Phi_0 \wedge \Phi_1$ iff
$\sigma, ins|\sigma', ins' \models \Phi_0$ and $\sigma, ins|\sigma', ins' \models \Phi_1$

Abbreviations (where sta projects the state of a configuration):

$$gs \mid \sigma, ins \models \Phi \ \hat{=}\ \mathsf{sta}(\mathsf{last}(gs)), \mathsf{inp}(gs) \mid \sigma, ins \models \Phi$$
$$gs \mid hs \models \Phi \ \hat{=}\ gs \mid \mathsf{sta}(\mathsf{last}(hs)), \mathsf{inp}(hs) \models \Phi$$

Next we define the release policy $\mathsf{RP}^\ell(gs)$ for a pre-run gs. Like knowledge, it is a set of input lists. This set of possibilities is what ℓ-observer is allowed to know, as specified by the ℓ-assumptions. Keep in mind that those are annotations with tag $\ell' \preceq \ell$. They typically have agreement formulas for other levels ℓ'' not visible to ℓ, since their purpose is to govern the flow of information from other levels to ℓ'.

Release policy of a pre-run RP^ℓ

$\mathsf{RP}^\ell([\langle\sigma_0\rangle]) = \{ins \mid ins \text{ is a list of input events}\}$

$\mathsf{RP}^\ell(gs \cdot g) = \{ins \mid ins \in \mathsf{RP}^\ell(gs)\}$
if $\mathsf{redex}(g)$ is not an ℓ-assumption

$\mathsf{RP}^\ell(gs \cdot g) = \{ins \mid ins \in \mathsf{RP}^\ell(gs) \wedge \exists \sigma.\ gs \cdot g|\sigma, ins \models \Phi\}$
if $\mathsf{redex}(g) = \mathsf{assume}^{\ell'}\ \Phi$ and $\ell' \preceq \ell$

The base case defines $\mathsf{RP}^\ell([\langle\sigma_0\rangle])$ to be the set of all input lists. The second clause says that if the next configuration g is not an assumption then the release policy is not changed by the step. The clause embodies the policy expressed by an ℓ-assumption Φ. It says that when the assumption is reached, the sequences ins deemed possible according to policy are those that were previously deemed possible and which in addition

[4] One could formulate knowledge of both states and inputs. That would capture things such as the fact that for the example (1), one can infer there was a state in which $y = 1$, after observation $[\mathsf{in}^P 0, \mathsf{out}^Q 1]$.

agree according to Φ with the state, $\mathsf{sta}(g)$, and inputs, of the current pre-run.

The release policy has no obvious connection with knowledge: a policy specification only restricts a few sensitive values whereas observation and logical omniscience about the program's semantics enables the observer to reason about what inputs possibly occurred.

The semantics of $\mathbb{A}\ell@n$ makes it true if either input list is shorter than n. The reason for this is to ensure that an assumption that is falsified by a minor pre-run cannot be made true by extending that pre-run.

Lemma 6. (a) $\mathsf{RP}^\ell(gs \cdot g) \subseteq \mathsf{RP}^\ell(gs)$, and (b) $ins \in \mathsf{RP}^\ell(gs)$ and $ins' \leq ins$ implies $ins' \in \mathsf{RP}^\ell(gs)$.

It is in the definition of security that the traces allowed by release policy are cut down to those compatible with program behavior, as expressed by $\mathsf{k}^\ell_{\rightarrow}$.

Secure pre-run

A pre-run gs is ***secure*** provided that for every prefix $hs{\cdot}g \leq gs$ (with hs nonempty), and every level ℓ, the step to g is secure at level ℓ. For the step to be secure at ℓ means that, for all ts, t with $hs \Downarrow ts$ and $hs \cdot g \Downarrow ts \cdot t$, if t is not in In^ℓ then

$$\mathsf{k}^\ell(ts \cdot t) \supseteq \mathsf{k}^\ell_{\rightarrow}(ts) \cap \mathsf{RP}^\ell(hs \cdot g)$$

Starting in Section V, security will be considered for programs in which ts, t are uniquely determined by gs (see Lemma 7). Always $\mathsf{k}^\ell(ts \cdot t) \subseteq \mathsf{k}^\ell_{\rightarrow}(ts)$ (Lemma 3), so the condition says that $\mathsf{k}^\ell(ts \cdot t)$ is no smaller—the learning no greater—than allowed by RP^ℓ.

The exclusion of visible input might seem to allow arbitrary learning at visible input steps. But nothing is learned at such steps. As per Lemma 5, the k^ℓ-set changes only by ruling out executions in which the ℓ-agent chose a different ℓ-input value. The interesting steps for security are visible outputs.

V. Small-step semantics for relational logic

In this section we define a notion called safety, adapted from Chudnov et al. [25], which relates a major pre-run to minor pre-runs with alternate input histories. This makes no reference to observations or release policy but instead directly interprets annotations in terms of aligned steps of the major and minor pre-run.

A. Well specified programs

Without loss of generality, we require that the program under consideration satisfies the following condition. A program is ***well specified*** provided that for every ℓ, with handler $\mathsf{input}^\ell x.c$, we have

- $c \equiv \mathsf{assume}^\ell\ \mathbb{A}x\,;d$, for some command d
- every output command $\mathsf{output}^{\ell'} e$ in c is immediately preceded in sequence by $\mathsf{assert}^{\ell'}\ \mathbb{A}e$

(We write $\equiv$ for syntactically identical.) Let us call these ***boilerplate annotations***. They help streamline the formalization in several ways, by encoding the baseline policy that is based on channel labels and the ordering on labels. For a concrete practical semantics one would make these annotations implicit. Explicit annotations would then be needed only for declassification (see the highlighted annotations in Figure. 1). Also, in practice there may be levels that need no handler because only the output channel is used, and multiple input channels for which the same handler is used.

The requirement for programs to be well specified loses no generality because the requisite annotations can be added without altering the behavior (except for adding $\bullet$ events, which are not observable).

Note that, owing to the program being well specified, RP^ℓ rules out input lists in which the visible inputs differ from those of the actual pre-run. Another technicality is the following, which captures why we do not need to bother tagging configurations with an indication of which handler is currently executing.[5]

Lemma 7. If $gs \Downarrow ts$ and $gs \Downarrow ts'$ then $ts = ts'$.

Proof. By induction on gs and cases on transition rules. For steps other than input, the argument is direct from determinacy for active configurations. For input, the argument depends on the program being well specified. Consider a step from $\langle\sigma\rangle$ to $\langle c, \sigma'\rangle$. By semantics, c is the handler body. Because the program is well specified, c has the form $\mathsf{assume}^\ell\ \mathbb{A}x; \ldots$, from which we have that the channel is ℓ and the input value was assigned to variable x. So the event is $\mathsf{in}^\ell n$ where n is $\sigma'(x)$. □

In light of Lemma 7, we can define the ***input history***, $\mathsf{inp}(gs)$, for any pre-run gs. It is simply $\mathsf{inp}(ts)$ where ts is the unique one with $gs \Downarrow ts$.

B. Safety

The definition of safety is based on some technical definitions concerning how one pre-run can be aligned with another.

Alignment, proper alignment for ℓ, coverage

For pre-runs gs, hs, an ***alignment*** from gs to hs is a relation $\alpha \subseteq dom(gs) \times dom(hs)$ that **(a)** is monotone, i.e., $\forall i, j, k, l$ with $i\alpha j$ and $k\alpha l$, $i < k \Rightarrow j \leq l$ and $j < l \Rightarrow i \leq k$; and **(b)** has prefix-closed domain and range, i.e., $i \in dom(\alpha)$ (resp. $rng(\alpha)$) and $0 \leq j < i$ imply $j \in dom(\alpha)$ (resp. $rng(\alpha)$).

A ***proper alignment for*** ℓ from gs to hs is an alignment α such that for all i, j with $i\alpha j$, if either $\mathsf{redex}(gs_i)$ or $\mathsf{redex}(hs_j)$ is an ℓ-annotation then $\mathsf{redex}(gs_i) = \mathsf{redex}(hs_j)$.

For α to ***cover*** the major pre-run gs (resp. the minor pre-run hs) means that $dom(\alpha) = dom(gs)$ (resp. $rng(\alpha) = dom(hs)$).

An ***ℓ-aligned pre-run pair*** is a triple (gs, hs, α) where α is a proper alignment from gs to hs for ℓ

The condition $\mathsf{redex}(gs_i) = \mathsf{redex}(hs_j)$ is meant to express that gs_i and hs_j are both active configurations, with exactly

[5]This is just an artifact of our parsimonious formalization: one could as well keep the input history in the program configuration.

the same assumption or assertion: not just the same level and formula but exactly the same occurrence in the program text, i.e., point in control flow. To make that precise one can provide each annotation occurrence with a unique identifying label (for which see Footnote 2).

In [25], the concern is to account for all possible initial states. Here we need to account for all possible input lists and associated pre-runs. This account is based on a classification: a minor pre-run may diverge or violate an assumption, in which case it can be disregarded—we call those 'fiats'. Alternatively, it may be in conformance with policy, or violate policy due to a mis-aligned annotation (called alignment failure) or due to an assertion failure. The idea is that a computation gs is safe if there are no alternate pre-runs resulting in assertion or alignment failure. Assumption fiat generalizes preconditions, removing from considerations pre-runs not relevant to policy. Divergence fiat is considered benign, an accord with the progress-insensitive security property.[6]

Classification of aligned pre-run pairs

The following are defined ***for level*** ℓ ***and for input list*** ins.

A ***conformance*** is an ℓ-aligned pre-run pair (gs, hs, α) where $\mathsf{inp}(hs) \leq ins$, α covers hs, and for all i, j, if $i\alpha j$ and $\mathsf{redex}(gs_i)$ is an ℓ-assertion or ℓ-assumption Φ then

$$gs\lfloor(i+1) \mid hs\lfloor(j+1) \models \Phi \tag{2}$$

Moreover either α covers gs or $\mathsf{inp}(hs) = ins$ and $\mathsf{last}(hs)$ is receptive.

An ***assumption fiat*** is ℓ-aligned (gs, hs, α) where $\mathsf{inp}(hs) \leq ins$, α covers hs, and there are i, j, Φ such that $i < |gs|$, $j = |hs| - 1$, $i\alpha j$, $\mathsf{redex}(gs_i)$ is $\mathbf{assume}^{\ell'}\,\Phi$ with $\ell' \preceq \ell$, Eqn. (2) does not hold, and $(gs\lfloor i,\ hs\lfloor j,\ \beta)$ is a conformance for ℓ, ins where $\beta = \{(k,l) \mid k\alpha l \wedge k < i \wedge l < j\}$.

An ***assertion failure*** is the same as an assumption fiat, except that $\mathsf{redex}(gs_i)$ is $\mathbf{assert}^{\ell'}\,\Phi$ with $\ell' \preceq \ell$.

An ***alignment failure*** is ℓ-aligned (gs, hs, α) where $\mathsf{inp}(hs) \leq ins$, α covers hs, and there are i, j, $i < |gs|$, $j = |hs| - 1$, $i\alpha j$, such that $\mathsf{redex}(gs_i)$ is an annotation command for ℓ, and $\mathsf{redex}(hs_j)$ is an ℓ-annotation different from $\mathsf{redex}(gs_i)$.

A ***divergence fiat*** is (gs, hs, α) where $\mathsf{inp}(hs) \leq ins$, α covers hs, and there is i, $i < |gs|$, such that $\mathsf{redex}(gs_i)$ is an ℓ-annotation command, $(gs\lfloor i, hs, \alpha)$ is a conformance for ℓ, ins, and for every $n \geq |hs|$ there is $hs' \geq hs$ of length n, such that $\mathsf{inp}(hs') = \mathsf{inp}(hs)$ and $\mathsf{last}(hs')$ is active and $\mathsf{redex}(\mathsf{last}(hs'))$ is not an ℓ-annotation.

Note that in the conformance condition (2) we evaluate an assertion or assumption in the aligned states together with the input lists. Similarly in the conditions for assumption fiat and assertion failure.

A subtlety in the definition of conformance has to do with the intention to account for all steps of gs. If ins provides enough inputs, this can be achieved, in which case α covers gs. If ins fails to provide enough inputs for the minor pre-run to fully align with gs, nonetheless hs should be as long as possible, i.e., reach a receptive configuration. The case where gs is not covered is not a satisfactory account of the safety of gs, but this is not a problem because the safety condition (below) quantifies over all input lists ins.

Divergence fiat needs to be understood in connection with programs being well specified and input total. Because ℓ-visible inputs and outputs are accompanied by annotations for ℓ, a divergence fiat indicates that the minor pre-run is continuing without producing visible output or consuming visible input. That could be due to a nonterminating loop with no outputs, or one with non-visible outputs. By contrast, in an alignment failure, the minor pre-run definitely reaches another ℓ-annotation, but one that does not match gs.[7]

For example, suppose the input history of the major pre-run gs is $[\mathsf{in}^{\ell}1, \mathsf{in}^{\ell}2]$. At level ℓ, with $ins = [\mathsf{in}^{\ell}3, \mathsf{in}^{\ell}2]$, we get an assumption fiat regardless of the program. That is because a well specified ℓ-handler begins with an assumption so any minor pre-run aligns properly and the initial agreement on input is false.

Safe pre-run

A pre-run gs is ***safe for*** ℓ ***and inputs*** ins iff there are hs, α such that (gs, hs, α) is a conformance, an assumption fiat, or a divergence fiat for ℓ, ins. A pre-run is ***safe for*** ℓ if for every ins it is safe for ℓ, ins. It is ***safe*** if it is safe for every ℓ.

Given that input sequences determine pre-runs (Lemma 2), one can define a safe input list to be one with safe pre-runs. We do not use that notion, but we do use some results that are used to connect safety with security. The first says extending a failed pre-run yields the same. The second says nonempty prefixes of a safe pre-run are safe.

Lemma 8. [forward fiat and failure] If $gs \cdot g$ and gs are pre-runs and (gs, hs, α) is an assumption or divergence fiat for ℓ, ins, or an assertion or alignment failure for ℓ, ins, then so is $(gs \cdot g, hs, \alpha)$.

Lemma 9. [backward safety] If $gs \cdot g$ and gs are pre-runs and $gs \cdot g$ is safe then so is gs.

The details in the definitions of failures, fiats, and conformance are motivated by the need to make these conditions mutually exclusive and exhaustive, as confirmed by the following.

[6] Whereas in [25] assumption and divergence fiats have minor pre-runs ('traces' in [25]) that end at the point of assumption violation or divergence, here our formulation allows that the minor pre-run continues beyond that point. Some details are also changed in accord with our treatment of annotations being checked in the step where they become the redex, rather than the following step.

[7] In addition to modifying the definitions of [25] to fit the reactive program model, we have also tightened the definition of divergence failure to make it mutually exclusive from alignment failure.

Lemma 10. [classification] For any gs, ins, and ℓ there are hs and α such that (gs, hs, α) is either a conformance, an assertion or alignment failure, or an assumption or divergence fiat for ℓ, ins.

The proof goes by induction on gs. Details in the classification of aligned pre-run pairs are motivated by details in the proof, which is provided as an appendix. The construction of hs and α amounts to the design of an ideal monitor. Simultaneously for the minor pre-runs of all ins, the monitor tracks the steps of the major run and reasons about the minor run for ins. If any ins results in an assertion or alignment failure, then continuing execution of gs is unsafe. We return to this in Section VII which discusses enforcement of safety.

VI. Relational safety implies epistemic security

Security says that for each observer level ℓ, what they may learn is within what is allowed by the release policy as specified by ℓ-annotations. Safety implies security.

Theorem 11. A pre-run that is safe for ℓ is secure for ℓ.

Proof sketch. Consider any ℓ. In accord with the definitions, we consider an arbitrary pre-run and go by induction on it.

The base case is the shortest pre-run, $[\langle\sigma_0\rangle]$, which is secure by definition: it has no prefixes $hs \cdot g$ with hs nonempty.

For the induction step, consider a pre-run $gs \cdot g$ that is safe. Safe pre-runs are prefix closed (Lemma 9), so gs is safe, hence gs is secure by induction hypothesis. So to prove $gs{\cdot}g$ is secure it remains to consider the last step. We must show

$$\mathsf{k}^\ell(ts \cdot t) \supseteq \mathsf{k}^\ell_{\rightarrow}(ts) \cap \mathsf{RP}^\ell(gs \cdot g) \quad \text{if } t \notin \mathsf{In}^\ell \tag{3}$$

for the unique ts, t that satisfy

$$gs \Downarrow ts \qquad gs \cdot g \Downarrow ts \cdot t \tag{4}$$

By Lemma 3, we have $\mathsf{k}^\ell(ts{\cdot}t) = \mathsf{k}^\ell_{\rightarrow}(ts)$ unless t is ℓ-visible, and this proves (3) for all transitions except a visible output $t \in \mathsf{Out}^\ell$.

To prove (3) for $t \in \mathsf{Out}^\ell$, consider any list of inputs ins. By safety of $gs{\cdot}g$ at ℓ, we have fs, α such that $(gs{\cdot}g, fs, \alpha)$ is a conformance, assumption fiat, or divergence fiat for ℓ, ins. We must prove that $ins \in \mathsf{k}^\ell_{\rightarrow}(ts)$ and $ins \in \mathsf{RP}^\ell(gs{\cdot}g)$ imply $ins \in \mathsf{k}^\ell(ts \cdot t)$, either by refuting one of the antecedents or by showing the consequent.

By definitions, $ins \in \mathsf{k}^\ell_{\rightarrow}(ts)$ means there are hs, us and $u \in \mathsf{Ev}^\ell$ with

$$ins = \mathsf{inp}(us) \qquad hs \Downarrow us \cdot u \qquad \mathsf{vis}^\ell(us) = \mathsf{vis}^\ell(ts) \tag{5}$$

Also, $ins \in \mathsf{k}^\ell(ts \cdot t)$ means there are fs', vs such that

$$ins = \mathsf{inp}(vs) \qquad fs' \Downarrow vs \qquad \mathsf{vis}^\ell(ts \cdot t) = \mathsf{vis}^\ell(vs) \tag{6}$$

There are quite a number of variables in play so let us review the situation. The major pre-run $gs{\cdot}g$ has trace $ts{\cdot}t$. The minor pre-run hs, with trace $us \cdot u$, witnesses that inputs ins were possible according to prior knowledge, because $\mathsf{vis}^\ell(us) = \mathsf{vis}^\ell(ts)$. Safety of $gs \cdot g$ accounts for the inputs ins by a pre-run fs for them that is either in conformance or is ruled out by assumption or divergence fiat.

We complete the proof by cases on whether $(gs{\cdot}g, fs, \alpha)$ is a conformance, assumption fiat, or divergence fiat. Conformance lets us derive fs' such that (6) holds. Assumption fiat lets us refute the policy hypothesis $ins \in \mathsf{RP}^\ell(gs \cdot g)$. Divergence fiat refutes the progress hypothesis $ins \in \mathsf{k}^\ell_{\rightarrow}(ts)$.

Case conformance: Suppose $(gs \cdot g, fs, \alpha)$ is a conformance for ℓ, ins. Because the program is well specified, from $t \in \mathsf{Out}^\ell$ and (4) we get that $\mathsf{redex}(\mathsf{last}(gs)) = \mathsf{output}^\ell e$ for some expression e, the preceding redex is skip, and the one before that has redex $\mathsf{assert}^\ell \mathbb{A} e$. Call the latter configuration gs_i. By proper alignment there is j with $i\alpha j$ and fs_j is the same assertion. By conformance, the assertion holds, so gs_i and fs_j agree on the value of e and thus the output t. Now, conformance allows that fs may not have reached the actual output command, but because the program is well specified it is going to do so. So we may take fs' to be either fs or the extension of fs with another step or two to reach the point of output. It is also possible that fs already took some steps beyond the output; these steps do not consume further input or produce ℓ-visible output, because $gs \cdot g$ does not do so, so in that case let $fs' = fs$. Either way, fs' satisfies (6) and we are done.

Case divergence fiat: Suppose $(gs{\cdot}g, fs, \alpha)$ is a divergence fiat for ℓ, ins. Suppose the divergence fiat is at i, j so that $((gs \cdot g)\lfloor i, fs\lfloor j, \alpha)$ is an ℓ-conformance but $\mathsf{redex}((gs \cdot g)_i)$ is an ℓ-annotation and neither fs_j nor any of its continuations reach an ℓ-annotation as redex. Thus pre-run $fs\lfloor j$ and its continuations are not doing further visible events following $\mathsf{last}(\mathsf{inp}(fs))$. Owing to $\mathsf{inp}(fs) \leq ins$ and determinacy (Lemma 2), if the progress condition (5) holds then the handler for $\mathsf{last}(\mathsf{inp}(fs))$ progresses at least as far as output b, after having produced all of the visible outputs of gs. But there cannot be in $gs{\cdot}g$ an annotation on which fs diverged, because its progress to an output would result in alignment failure not divergence failure. So divergence fiat refutes the progress hypothesis $ins \in \mathsf{k}^\ell_{\rightarrow}(ts)$.

Case assumption fiat: Suppose $(gs{\cdot}g, fs, \alpha)$ is an assumption fiat for ℓ, ins. Suppose the assumption fiat is at i, j so that $((gs \cdot g)\lfloor i, fs\lfloor j, \alpha)$ is a conformance and $\mathsf{redex}((gs \cdot g)_i)$ is an ℓ-assumption of formula Φ (and so is $\mathsf{redex}(fs_j)$) but $(gs \cdot g)\lfloor(i+1)|fs\lfloor(j+1) \not\models \Phi$. Let ins' be $\mathsf{inp}(fs\lfloor j)$. By definition of RP^ℓ we have $ins' \notin \mathsf{RP}^\ell((gs \cdot g)\lfloor i)$. By Lemma 6(a) we get $ins' \notin \mathsf{RP}^\ell(gs{\cdot}g)$ and then by Lemma 6(b) we get $ins \notin \mathsf{RP}^\ell(gs \cdot g)$. □

VII. Enforcement of safety

The knowledge based security property is proposed as an intuitively clear interpretation for policies expressed as program annotations. The safety property, on the other hand, seems less transparent but is designed as a bridge to effective assurance of security. This section sketches how that assurance can be provided by runtime monitoring of individual executions and by static analysis that verifies security of all executions.

A. Monitoring

The safety property of Section V is adapted from the work of Chudnov et al. [25] who proposed to view a monitor as performing an abstract interpretation to account for all the minor pre-runs vis a vis the major one. Their work takes safety as the security property, whereas we take safety to be a convenient basis for enforcement of a knowledge based security property. In this section we briefly explain their monitor and outline how it can be adapted to safety in the setting of our reactive language.

Chudnov et al. [25] describe an idealized monitor in terms of configurations augmented with the monitor state, which includes a set of relational formulas known to hold with respect to the minor pre-runs. As an intermediate stage in the constructive justification of an implementable monitor, monitoring is described for the property "safe with respect to particular inputs", like our "safe for ins". This property is stronger than safety, in part because it provides a relation at every aligned pair of steps. The monitor leverages those relations in order to check assertions and alignment, ensuring that it can detect possible violation of safety. A sound approximation of safety can be maintained by reasoning with the formulas and the known states of the major run, independent of the minor pre-run. Hence the monitor effectively reasons about all minor pre-runs simultaneously.

Following prior works on so-called hybrid monitors that leverage static analyses, in [25] the monitor state includes a conventional labelling of variables, as a representation of agreements known at the current aligned points. The monitor also maintains a stack, to track implicit flow: at an aligned pair of execution steps, the monitor can determine whether the minor pre-run is at the same control point, and if not then what path it is on. This allows to update the variable labelling and known formulas based on possible effects of "high branches" with differing control flow.

Chudnov et al. [25] claim their approach is a systematic way to derive monitors for richer languages. The monitor actions are defined hand in hand with proving that the monitor maintains the strengthened safety condition, in an argument refining one similar to our proof of Lemma 10. Assaf et al. [8] reformulate the ideas in terms of a compositional program semantics and Galois connections that account for the way relational formulas abstract from agreements. In these terms they derive monitors by calculation—inspired by the constructive derivation of abstract interpreters by Cousot [28]—and show how a monitor can leverage existing static analyses formulated as abstract interpretations. In [8], variable labellings are dispensed with and the control level stack is implicitly represented in the recursively defined interpreter. Their formulation seems less easy to connect with knowledge semantics, so here we sketch how the monitor of [25] can be adapted to the reactive language.

The language of [25] does not include handlers and outputs. But given that programs are well specified, the monitor does not need to do anything special about outputs: they are protected by assertions which the monitor checks. The monitor does need to keep track of input values, but these are assigned to variables, and the assumption at the start of a handler informs the monitor that the minor pre-runs under consideration are in agreement. As shown in [25], a significant amount of reasoning can be done simply based on evaluating assertions in the current pre-run state and maintaining known agreements.

There are two significant changes that need to be made to the monitor of [25]. The first change is to handle agreement formulas $\mathbb{A}\ell@n$ that refer to inputs. Roughly, the monitor needs to establish agreements on variables being output—for assertions—based on assumed agreements on inputs. One obvious principle is that if the monitor is maintaining a counter i of inputs on channel ℓ, it can assume at the beginning of the ℓ handler that the input variable x of the handler has the value $chan^{\ell}[i]$, and thus from the assumed agreement on x we infer $\mathbb{A}\ell@i$. Perhaps surprisingly, this does not require the monitor to store a list of inputs seen. Our formula language deliberately omits state predicates that refer directly to inputs (recall Lemma 6); if that is desired in a specification, one needs to store inputs in locals (e.g., of type list) in order to express conditions on them. In maintaining known formulas, the monitor needs to drop those that may have been falsified by minor execution along a different control path. An agreement like $\mathbb{A}S@b-p+15$ with an expression for its offset needs to be updated or discarded in accord with changes to b and p.

The second change is to handle multiple security levels. The obvious idea is to maintain, for each ℓ, monitor state for the known agreements and level stack for alignments for safety at level ℓ. For a small number of levels this could be fine, but a more sophisticated treatment is preferable in case of a large number. For example, agreement at ℓ implies agreement at ℓ' for $\ell' \preceq \ell$; this can be exploited by the reasoning component, to reduce the number of agreement formulas stored.

B. Relational logic

This section sketches how safety can be assured for all executions, using techniques and tools from relational verification, in particular relational Hoare logics (RHL). Although some RHLs are for probabilistic or other quantitative properties (e.g., [13]), most aim to provide verification of a simple 2-safety property (e.g., [15], [37]). To explain, let us use plain letters R, S, in addition to Φ, for relational formulas. In the simplest case, the judgment $\{R\}c \sim d\{S\}$ says that for any pair σ, τ of states related by R, if c terminates on σ and d on τ then the final states are related by S. (There are other variations: both terminate; or, if c terminates then so does d.) For security specification, R and S express agreements, and d is the same program as c. But the verification often requires the general form $c \sim d$ for different programs. The typical case is a "high conditional" where it cannot be ensured that both runs take the same branch so the alternatives need to be related somehow.

In richer languages, e.g., with pointers, the judgement also says there are no runtime errors. In the case of judgments

under hypotheses—where c and d can call procedures that have relational contracts—a modular correctness judgment says that procedures are never called in states outside their preconditions. We propose yet another variant, to cater for intermediate assertions and assumptions.

The alignment of intermediate computation steps is not explicit in most work on RHL, although it is explicit in some automatic verification techniques [33]. (For clarity we focus here on logics rather than verification tools, and in any case logics explicate the reasoning principles that underly tools.) Implicitly, however, the syntax directed rules of RHL designate alignments. For example, to prove $\{R\}c1; c2 \sim d1; d2\{S\}$ a standard rule is to prove $\{R\}c1 \sim d1\{Q\}$ and $\{Q\}c2 \sim d2\{S\}$ for some Q. Notice that in fact the two premises establish that a pair of runs of $c1; c2$ and $d1; d2$ can be aligned "at the semicolons" and Q holds at the alignment point. Here Q is asserted as postcondition for the first part, and assumed as precondition for the second. (Compare with the monitor, which needs relations at intermediate alignment points.) DN:[Elaborate here?]

Logics also need rules to relate differently structure code fragments, like alternate high branches, for which there are not useful intermediate alignment points.

Just as the monitor can reason that a formula Φ continues to hold unless variables in it get updated, in RHL one finds "frame rules" that infer $\{R \wedge Q\}c \sim d\{S \wedge Q\}$ from $\{R\}c \sim d\{S\}$ if Q depends on no location that is updated in c or d.

For our purposes, it is the relational annotations that are intended to be aligned in the two runs, so for reasoning about relational assumptions and assertions it suffices to consider $c \sim c$ where c is of the form $\mathsf{assume}^{\ell}\Phi$ or $\mathsf{assert}^{\ell}\Phi$. We ignore ℓ now and return to it later. The obvious axiom for assumptions is $\{R\}\mathsf{assume}\Phi \sim \mathsf{assume}\Phi\{R \wedge \Phi\}$. One rule for assertions is that $\{R\}\mathsf{assert}\Phi \sim \mathsf{assert}\Phi\{R\}$ holds if $R \Rightarrow \Phi$. To verify a reactive program, verify that $\{true\}c \sim c\{true\}$ for each handler body c.

It is beyond the scope of this paper to work out details, but we observe that most RHL rules are in fact sound with respect to the semantics we need for judgments, which slightly generalizes safety. Here is how we propose to interpret the judgment form $\{R\}c \sim d\{S\}$ at level ℓ. For any pre-run of c from any initial state, and for every input list ins, the pre-run of d in ins (and again from any initial state) yields a conformance, assumption fiat, or divergence fiat. For this purpose, only the ℓ annotations are relevant—the rules for assume and assert would only be used for ℓ-annotations and other annotations (for $\ell' \not\preceq \ell$) would be treated as skip. By contrast with the definitions of knowledge, progress knowledge, and safety, here we quantify not just over runs from the initial configuration but runs from arbitary configurations. And we align runs of c with runs of a possibly different program d, to validate the RHL rules for non-synchronized conditionals and loops.

By considering all starting states, the property is compositional in this sense: if $\{true\}c \sim c\{true\}$ is valid for every handler body c then the complete program is safe (in the sense of Section V).

Justifying a logic's rules with respect to this semantics involves reasoning about alignments, for which purpose the work of Banerjee et al. [10] is particularly suited because their logic provides forms closely related to relational assertions and assumptions: special "biprogram" constructs that are used to express syntactic alignments of code such as synchronized procedure calls and agreeing conditionals. The operational semantics of biprograms explicitly models step-by-step alignment of all intermediate points (as in a monitor) such that the requisite agreements hold. It should be possible to extend their semantics to include relational annotations.

In effect each level ℓ has an associated specification, the ℓ-annotations, to be verified. An obvious question is whether multiple verifications are needed or can the different levels—correcness with respect to observers at a level—be verified simultaneously. The use of RHLs for multi-level information flow has been considered in the context of SparkADA [1]. Because it is the same program being verified at each level, one can use level-indexed formulas and suitable entailments to combine the multiple verifications into a single one, though not necessarily reducing the overall effort compared with a separate verification for each of the many or few levels of interest.

VIII. Related work

Sabelfeld and Sands [41] systematized the early work on semantics and enforcement of declassification, classified declassification policies according to dimensions (*what-where-who-when*) and identified restrictions that prevent unintended disclosure of information. Much of the following work focused on the semantics and enforcement of policies across several dimensions.

A. Program logics and type systems for declassification

Mantel and Reinhard [35] enforce declassification policies in the the *where* and *what* dimensions. They identify two kinds of *what* properties, suitable for controlling timing leaks in concurrent programs, but possessing different characteristics. Enforcement is done with a type system.

Askarov and Sabelfeld [6] proposed a type system that allowed enforcing policies in the *where* and *what* dimensions. Their security condition, *delimited localized release*, fulfills the principles identified in [41]. Later they reformulated the security property (*gradual release*) in terms of attacker knowledge [3]; this has given rise to a whole line of work on epistemic information-flow security (Section VIII-C).

Barthe et al. [12] propose a modular type-and-effect system for enforcing declassification that combines both *what* and *where* dimensions in one construct. The security property is related to delimited localized release. Modularity of the type system allows straightforward proofs of typing preservation for security preserving compilation, as demonstrated for a Java-like language and JVM-like bytecode.

Banerjee at al. [11] propose a combination of a type system and a program logic to specify and enforce declassification policies that combine *what*, *where* and *dimensions*.

Enforcement is done by means of a type system that emits proof obligations in the form of relational and non-relational assertions.

A prudent requirement for declassification functions is to reduce information content of the result. Yet, even a well designed declassification function can be abused and applied to the same value several times, disclosing more information than intended. To address this problem Kaneko and Kobayashi [32] proposed a linear type system that limits how often a declassification function can be applied, as well as how often the declassified value can be used. This allows to bound the information disclosed by programs. This approach can be emulated using relational assertions only allow disclosure if a counter c is positive ($\mathbb{B}\, c > 0 \Rightarrow \mathbb{A}e$), and decrementing it after each disclosure.

B. Dynamic enforcement of declassification

Demongeot et al. extend the Le Guernic's [34] noninterference monitor with a declassification expression and show how to enforce delimited localized release dynamically.

Askarov and Sabelfeld [7] show how to enforce a generalization of gradual release and delimited localized release dynamically. Their I/O model is interactive and multi-channel, but in contrast to ours, their input operations are blocking. Other differences from our work include support of dynamic code evaluation and both termination-insensitive and sensitive formulations of the monitor.

Jina et al. [31] extend the inlined monitor from [26] with a declassification primitive. They give a high level proof sketch that the monitor enforces a knowledge-based security property that appears similar to gradual release [3].

C. Release policies described in terms of knowledge

As pointed out in [9] and emphasized in [22], a wide range of policies for downgrading can be described in terms of attacker knowledge. Each step of a program can produce an observation that the attacker can learn something about the inputs or the initial memory of the program [3]. A release policy [11], [7] bounds the learning.

Besson et al. [17] devise monitors that directly approximate the attacker's knowledge, which might used to enforce an epistemically specified policy. Their work focuses on improving permissiveness for enforcement of noninterference.

D. Dynamic policies

Dynamic information flow policies have recently emerged as an alternative to downgrading. Instead of specifying the values to be released, the programmers are allowed to change the policies—usually, by changing the partial ordering of security levels.

Broberg and Sands have suggested and evolved the approach of labels conditioned on program state [19] and using it to encode declassification expressions [20] and, finally, generalizing them to a role-based model [21].

Swamy and Hicks [43] propose to specify stateful declassification policies, motivated by DoD rules, using automata and use a sophisticated type system to enforce them in a functional language. The approach can be implemented with our approach using relational assertions and assumptions, additional program variables, representing the state of the policy automaton, and program instrumentation, implementing its transition rules. The state variables would be used as guards for agreements on released data in relational assumptions.

Askarov and Chong [4] address policies that involve events or conditions by using a flow ordering that is mutable. Their epistemic semantics says that in each step, what is learned is allowed by the current flow relation; this is a direct precursor to our notion of release policy.

Broberg at al [22] classify dynamic policies, also giving semantics to them in terms of attacker knowledge. They identify *facets* of policies from previous work: termination sensitivity, time-transitivity, flow replaying, direct release and flow whitelisting. Like Askarov and Chong [4], they suggest a simple approach for encoding policy changes in the program, using $A \rightarrow B$ to enable a flow from A to B and $A \nrightarrow B$ to disable it. Finally, they relate dynamic policies to declassification. Since the set of labels is fixed—only their ordering changes—we can encode such policies in our system, albeit verbosely. Consider two levels A and B and a boolean variable ab. Then the simple dynamic policy program [22] $A \rightarrow B;\ b := a;\ A \nrightarrow B$ can be expressed as $ab := 1;\ b := a;\ \mathsf{assume}^B \mathbb{B}ab \Rightarrow \mathbb{A}b;\ ab := 0$ by representing the flow relations in the program state itself and using them as agreement conditions.

Buiras and van Delft [23] extend the dynamic information flow tracking library `LIO` [42] with the ability to change label ordering at run-time. The property that the library enforces is noninterference with respect to the policy in the current state as well as the absence of flows due to the change in policy itself.

Arden and Myers [2] investigate the how dynamic authorization and information flow policies affect each other. They formalize their approach as a program logic for a small functional language for specifying decentralized authorization protocols and use it to verify two protocols. This work is a basis for the investigation by Ceccetti et al. [24] of a new interaction between integrity and confidentiality, specifically a form of endorsement that is dual to robust declassification.

Bauereiß et al. [14] implement a distributed social media platform with dynamic information flow policies formulated in terms of trigger conditions and bounds on release. They verify the system's security by interactive theorem proving.

E. Information-flow security for interactive programs

O'Neill et al. [39] study the security properties of interactive programs and devise strategies that the attacker can use to obtain more information than intended. However, Clark and Hunt [27] determined that strategies give the attacker no advantage in case of deterministic programs. This justifies works like [4] in which program semantics is formalized in terms of an initially given complete input stream. We mention

that particular work because it appears their notion of fine-grained policies, though formulated in a way very similar to our notion of secure pre-run, may admit clairvoyant policies. By contrast, this cannot happen in our program semantics because future inputs are not given in advance.

Vanhoef et al. [44] propose a declassification approach for event-driven programs that relies on what we call *policy state*, which is updated in response to input events. This makes possible an extensional policy formulation: at each event, the policy says what function of the event value may be released, depending on what kind of event and depending on the policy state. In addition, policy specifies how the policy state transitions in response to input events. The security directly is not formulated in terms of knowledge, but in effect it is similar to that in [9]. Their approach can be implemented by instrumenting the program with a variable to store policy state, updated upon inputs, together with assumed agreements conditioned on predicates over the state.

F. Release policies as program annotations

Balliu et al. [9] remark that a wide range of downgrading policies in a small fragment of epistemic temporal logic. We aim to show in effect that correctness for this fragment can be expressed by the very simple means of relational assertions and assertions, as anticipated by [11].

Vaughan and Chong [45] enforce policies conditioned on certain events (whether specific methods have ever been invoked) and referring to inputs by indexing input streams backwards, enabling references n-th most recent input on a given channel.

An attractive idea in [44] is to use declassification statements not as a means to specify policy, but only as a means to inform the enforcement mechanism (in their case, *secure multi-execution* [29]) what is the intended policy. One can imagine other heuristic or methodological means for manually or automatically (e.g. [45], [40]) annotating a program, at runtime or prior to runtime, but that is not our concern here. Rather, we are taking the prior step, which is to give a semantics for annotations and show how to express various sorts of policy using annotations, so that their semantics is in accord with the original proposals.

IX. Conclusion

The suitability of relational program logic to reason about information flow has been known for some time and continues to bear fruit (e.g., [15], [36]). Relational annotations are less explored and their prior use for monitoring [25], [8] involved intricate and unusual semantics. The use of annotations to specify declassification is attractive, in part because annotations that refer to program state may be relatively easy for programmers to become familiar with and integrate into software interfaces and practices, by contrast with specialized notations for security. Relational assumptions are the least familiar but our new knowledge semantics justifies an intuitive reading: an assumed agreement on some expression, referring to program state or directly to past inputs, means "the observer at that level is now permitted to know this". We suggest that while other specialized notations may be convenient in particular settings, they may be explained and enforced by desugaring to relational annotations as presented here. Besides using knowledge to justify the more intricate "safety semantics", we have taken first steps towards showing that the later is not only useful for monitoring but also for static verification in relational program logic.

Formulating security policy using assertions and assumptions (and relational contracts for procedures) enables sophisticated policies that depend on whatever conditions are manifest in the program code, including instrumentation provided in support of policy specification. For example, database privacy based on the notion of differential privacy may involve dynamic tracking of prior queries in order to determine whether a candidate query risks violating a privacy budget, and key management may involve statistics about session duration and current risk profile. Such policies ultimately need justification in terms of quantitative analysis of the information flows and the environment. Good high level policies may make reference to dynamic conditions such as prior queries, session duration, retry count, and the exchange rate of Bitcoin. However, verifying or dynamically enforcing that the implementation conforms to policy is based on the code and its interfaces. Such quantitative conditions can be expressed in terms of ordinary assertions about program state and reasoning may well benefit from integration with ordinary program invariants and contracts.

Such considerations motivate further exploration of the knowledge semantics, for example, to include the current state in the knowledge rather than existentially quantifying it in the release policy (which we did here for simplicity). The semantics here is not specific to the programming language and can easily be extended to richer features such as procedures and the heap, within a sequential execution model. To extend the semantics to encompass integrity requirements, a key starting point is the knowledge based approach of Askarov and Myers [5] (see also [24]). Extension to concurrency is challenging and will necessitate dealing with nondeterminacy, which is a significant complication for relational properties. However, assertion based methods underlie most reasoning systems for concurrency in programs, in particular providing methodologies by which reasoning can be done largely in sequential style owing to clear interaction points (locking, wait-free atomics).

Acknowledgement

The views expressed in this paper are those of the authors and not D. E. Shaw & Co., L.P. or any of its affiliates. The first author thanks Galois, Inc. for their support while employed there. The second author was partially supported by NSF awards CCF-1649884, CNS-1718713, and CCF-1521602. Discussions with Mounir Assaf were very helpful, as were the CSF reviews.

REFERENCES

[1] T. Amtoft, J. Hatcliff, E. Rodríguez, Robby, J. Hoag, and D. Greve, "Specification and checking of software contracts for conditional information flow," in *Formal Methods*, ser. LNCS, vol. 5014, 2008.

[2] O. Arden and A. C. Myers, "A calculus for flow-limited authorization," in *IEEE Computer Security Foundations Symposium*, 2016, pp. 135–149.

[3] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *IEEE Symposium on Security and Privacy*, 2007.

[4] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *IEEE Computer Security Foundations Symposium*, 2012, pp. 308–322.

[5] A. Askarov and A. C. Myers, "Attacker control and impact for confidentiality and integrity," *Logical Methods in Computer Science*, vol. 7, no. 3, 2011.

[6] A. Askarov and A. Sabelfeld, "Localized delimited release: combining the what and where dimensions of information release," in *ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2007, pp. 53–60.

[7] ——, "Tight enforcement of information-release policies for dynamic languages," *IEEE Computer Security Foundations Symposium*, 2009.

[8] M. Assaf and D. A. Naumann, "Calculational design of information flow monitors," in *IEEE Computer Security Foundations Symposium*, 2016, pp. 210–224.

[9] M. Balliu, M. Dam, and G. Le Guernic, "Epistemic temporal logic for information flow security," in *ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2011, p. 6.

[10] A. Banerjee, D. A. Naumann, and M. Nikouei, "Relational logic with framing and hypotheses," in *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. LIPIcs, vol. 65, 2016, pp. 11:1–11:16.

[11] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *IEEE Symposium on Security and Privacy*, 2008.

[12] G. Barthe, S. Cavadini, and T. Rezk, "Tractable enforcement of declassification policies," in *IEEE Computer Security Foundations Symposium*, 2008, pp. 83–97.

[13] G. Barthe, T. Espitau, B. Grégoire, J. Hsu, and P. Strub, "Proving expected sensitivity of probabilistic programs," *PACMPL*, vol. 2, no. POPL, pp. 57:1–57:29, 2018.

[14] T. Bauereiß, A. P. Gritti, A. Popescu, and F. Raimondi, "CoSMeDis: a distributed social media platform with formally verified confidentiality guarantees," in *IEEE Symposium on Security and Privacy*, 2017, pp. 729–748.

[15] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *ACM Symposium on Principles of Programming Languages*, 2004.

[16] L. Beringer, "End-to-end multilevel hybrid information flow control," in *Asian Symposium on Programming Languages and Systems*, ser. LNCS, 2012, vol. 7705.

[17] F. Besson, N. Bielova, and T. P. Jensen, "Hybrid monitoring of attacker knowledge," in *IEEE Computer Security Foundations Symposium*, 2016, pp. 225–238.

[18] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *ACM Computer and Communications Security*, 2009, pp. 79–90.

[19] N. Broberg and D. Sands, "Flow locks," in *European Symposium on Programming*, ser. LNCS, vol. 3924, 2006.

[20] ——, "Flow-sensitive semantics for dynamic information flow policies," in *ACM Workshop on Programming Languages and Analysis for Security*, 2009.

[21] ——, "Paralocks – role-based information flow control and beyond," in *ACM Symposium on Principles of Programming Languages*, 2010.

[22] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *IEEE Computer Security Foundations Symposium*, 2015, pp. 122–136.

[23] P. Buiras and B. van Delft, "Dynamic enforcement of dynamic policies," in *ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2015, pp. 28–41.

[24] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *ACM Computer and Communications Security*, 2017, pp. 1875–1891.

[25] A. Chudnov, G. Kuan, and D. A. Naumann, "Information flow monitoring as abstract interpretation for relational logic," in *IEEE Computer Security Foundations Symposium*, 2014, pp. 48–62.

[26] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *IEEE Computer Security Foundations Symposium*, 2010, pp. 200–214.

[27] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *Formal Aspects in Security and Trust*, ser. LNCS, vol. 5491, 2008.

[28] P. Cousot, "The calculational design of a generic abstract interpreter," in *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. NATO ASI Series F. IOS Press, Amsterdam, 1999, vol. 173, pp. 421–506.

[29] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *IEEE Symposium on Security and Privacy*, 2010.

[30] C. Dwork, A. Roth *et al.*, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.

[31] L. Jina, H. Zhub, and H. Mab, "Monitor inlining for declassification policy," *Journal of Information & Computational Science*, vol. 12, no. 12, pp. 4697–4704, 2015.

[32] Y. Kaneko and N. Kobayashi, "Linear declassification," in *European Symposium on Programming*. Springer, 2008, pp. 224–238.

[33] M. Kovács, H. Seidl, and B. Finkbeiner, "Relational abstract interpretation for the verification of 2-hypersafety properties," in *ACM Conference on Computer and Communications Security*, 2013.

[34] G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," in *Advances in Computer Science: Secure Software and Related Issues, 11th Asian Computing Science Conference 2006 (Revised Selected Papers)*, ser. LNCS, vol. 4435, 2008.

[35] H. Mantel and A. Reinhard, "Controlling the what and where of declassification in language-based security," in *European Symposium on Programming*. Springer, 2007, pp. 141–156.

[36] C. Müller, M. Kovács, and H. Seidl, "An analysis of universal information flow based on self-composition," in *IEEE Computer Security Foundations Symposium*, 2015, pp. 380–393.

[37] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *IEEE Symposium on Security and Privacy*, 2011.

[38] ——, "Dependent type theory for verification of information flow and access control policies," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 2, 2013.

[39] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Computer Security Foundations Workshop, 2006. 19th IEEE*. IEEE, 2006, pp. 12–pp.

[40] B. P. Rocha, S. Bandhakavi, J. den Hartog, W. H. Winsborough, and S. Etalle, "Towards static flow-based declassification for legacy and untrusted programs," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010, pp. 93–108.

[41] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," *Journal of Computer Security*, 2007.

[42] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *ACM Sigplan Notices: Haskell Symposium*, vol. 46, no. 12. ACM, 2011, pp. 95–106.

[43] N. Swamy and M. Hicks, "Verified enforcement of stateful information release policies," *ACM Sigplan Notices*, vol. 43, no. 12, pp. 21–31, 2009.

[44] M. Vanhoef, W. D. Groef, D. Devriese, F. Piessens, and T. Rezk, "Stateful declassification policies for event-driven programs," in *IEEE Computer Security Foundations Symposium*, 2014, pp. 293–307.

[45] J. A. Vaughan and S. Chong, "Inference of expressive declassification policies," in *IEEE Symposium on Security and Privacy*, 2011, pp. 180–195.

APPENDIX

A. Proof of Lemma 2

By induction on ins, with (a)–(c) as induction hypothesis. In the base case, $ins = [\,]$ and gs is $[\langle\sigma_0\rangle]$. In the induction case, suppose ins is $ins' \cdot t$ and gs satisfies (a)–(c) for ins'.

If $\mathsf{inp}(gs) = ins'$ then all inputs were handled and a receptive configuration was reached. So gs can be extended by further transitions starting with the input of a to its handler. If the handler diverges, then gs itself satisfies (a)–(c) so we are

done; otherwise it can be run to completion, which results in $gs \cdot gs'$ where $\mathsf{last}(gs')$ is receptive and $\mathsf{inp}(gs \cdot gs') = ins' \cdot t$.

If $\mathsf{inp}(gs) < ins'$ then an earlier input is already divergent so by induction hypothesis we get (a) and (b) for $ins \cdot t$.

B. Proof of Lemma 6

(a) Direct from definition of RP^ℓ. (b) By induction on gs. In the base case (the initial configuration) the antecedent is false because no ins is rejected. For the induction step, suppose $ins \notin \mathsf{RP}^\ell(gs \cdot g)$. This can be because $ins \notin \mathsf{RP}^\ell(gs)$ or because g is an assumption that does not hold for ins. If $ins \notin \mathsf{RP}^\ell(gs)$ then by induction $ins' \notin \mathsf{RP}^\ell(gs)$ whence $ins' \notin \mathsf{RP}^\ell(gs \cdot g)$ by (a). If an assumption does not hold for ins, we rely on the property of relational formula semantics that if a formula is false for two states and two input lists then it is also false for extensions of one or both of those lists. That property can be proved by induction on formulas. It relies on the formula language not providing direct means to refer to the length of the input lists, treating the form $\mathbb{A}\ell@n$ as true for inputs that are too short, and not allowing agreements under negation.

C. Proof of Lemma 10

Consider any ℓ and ins, and go by induction on gs.

In the base case, gs is $[\langle\sigma_0\rangle]$; we take $hs := [\langle\sigma_0\rangle]$ and $\alpha := \{(0,0\}$. This forms a conformance.

In the induction case, gs has the form $fs \cdot g$ and by induction we have hs, α with (fs, hs, α) a conformance, fiat, or failure for ℓ, ins. If it is a fiat or failure then so is $(fs \cdot g, hs, \alpha)$, by Lemma 8, and we are done. If ins has been exhausted, i.e., (fs, hs, α) is a conformance such that $\mathsf{inp}(hs) = ins$ and $\mathsf{last}(hs)$ is receptive, then $(fs \cdot g, hs, \alpha)$ is a conformance and we are done.

It remains to consider the case that (fs, hs, α) is a conformance such that α covers fs and either $\mathsf{inp}(hs) < ins$ or $\mathsf{last}(hs)$ is active. We proceed by cases of $\mathsf{last}(fs)$.

- If $\mathsf{last}(fs)$ is receptive and the transition to g is for a visible input on some $\ell' \preceq \ell$, so $\mathsf{redex}(g)$ is $\mathsf{assume}^{\ell'}\ \mathbb{A}x$ (for some x) then construct $hs' \geq hs$ by successive steps, maintaining $\mathsf{inp}(hs') \leq ins$ (recall Lemma 2), until $\mathsf{redex}(\mathsf{last}(hs'))$ is an ℓ-annotation.
  - If this is not possible because $\mathsf{inp}(hs') = ins$ and $\mathsf{last}(hs')$ is receptive, then $(fs \cdot g, hs', \beta)$ is a conformance, where $\beta = \alpha \cup \{(|fs| - 1, j) \mid |hs| \leq j < |hs'|\}$.
  - If an ℓ-annotation is reached at $\mathsf{last}(hs')$ then there are three sub-cases:
    - If the annotation $\mathsf{redex}(\mathsf{last}(hs'))$ is the same as $\mathsf{redex}(g)$, and $\mathsf{sta}(g), \mathsf{sta}(\mathsf{last}(hs'))$ agree on x, then $(fs \cdot g, hs', \gamma)$ is a conformance, where $\gamma = \alpha \cup \{(|fs| - 1, j) \mid |hs| \leq j|hs'| - 1\} \cup \{(|fs|, |hs'| - 1)\}$.
    - If the annotation is the same as $\mathsf{redex}(g)$ but the states do not agree on x, then $(fs \cdot g, hs', \delta)$ is an assumption fiat, where $\delta = \alpha \cup \{(|fs| - 1, j) \mid |hs| \leq j|hs'| - 1\}$.
    - If $\mathsf{redex}(\mathsf{last}(hs'))$ differs from $\mathsf{redex}(g)$ then $(fs \cdot g, hs', \delta)$ is an alignment failure, where δ is as in the preceding.
  - If neither ins is exhausted nor an ℓ-annotation reached while growing hs', then the minor pre-run is diverging without doing further visible input or output. So $(fs \cdot g, hs, \alpha)$ is a divergence fiat.
- If $\mathsf{last}(fs)$ is receptive and the transition to g is for a non-visible input, so $\mathsf{redex}(g)$ is an assumed agreement for some $\ell' \not\preceq \ell$, then $(fs \cdot g, hs, \alpha \cup \{(|fs|, |hs| - 1)\})$ is a conformance. (Here we use that $\mathsf{last}(fs)$ cannot be an annotation, so neither can $\mathsf{last}(hs)$.)
- If $\mathsf{last}(fs)$ is active, we have these sub-cases:
  - If $\mathsf{redex}(g)$ is not an ℓ-annotation then $(fs \cdot g, hs, \alpha \cup \{(|fs|, |hs| - 1)\})$ is a conformance.
  - If $\mathsf{redex}(g)$ is an ℓ-annotation then construct $hs' \geq hs$ by successive steps, maintaining $\mathsf{inp}(hs') \leq ins$ until $\mathsf{redex}(\mathsf{last}(hs'))$ is an ℓ-annotation.
    - If this is not possible because $\mathsf{inp}(hs') = ins$ and $\mathsf{last}(hs')$ is receptive, then $(fs \cdot g, hs', \beta)$ is a conformance, where β is defined like earlier in this proof.
    - If an ℓ-annotation is reached, but $\mathsf{redex}(\mathsf{last}(hs'))$ is different from $\mathsf{redex}(g)$, then we obtain an alignment failure.
    - If $\mathsf{redex}(\mathsf{last}(hs'))$ is the same as $\mathsf{redex}(g)$, i.e., an assumption or assertion at some level $\ell' \preceq \ell$ with formula Φ that is satisfied by $fs \cdot g, hs'$, then $(fs \cdot g, hs', \gamma)$ is a conformance, where $\gamma = \beta \cup \{(|fs|, |hs'| - 1)\}$.
    - If $\mathsf{redex}(\mathsf{last}(hs'))$ is the same as $\mathsf{redex}(g)$ but the formula does not hold, we obtain an assumption fiat or assertion failure.
    - If none of the above apply, then we obtain divergence fiat.