Information Flow Monitoring as Abstract Interpretation for Relational Logic

Andrey Chudnov
Stevens Institute of Technology

George Kuan
HRL Laboratories, LLC

David A. Naumann
Stevens Institute of Technology

***Abstract*—A number of systems have been developed for dynamic information flow control (IFC). In such systems, the security policy is expressed by labeling input and output channels; it is enforced by tracking and checking labels on data. Systems have been proven to enforce some form of noninterference (NI), formalized as a property of two runs of the program. In practice, NI is too strong and it is desirable to enforce some relaxation of NI that allows downgrading under constraints that have been classified as 'what', 'where', 'who', or 'when' policies. To encompass a broad range of policies, relational logic has been proposed as a means to specify and statically enforce policy. This paper shows how relational logic policies can be dynamically checked. To do so, we provide a new account of monitoring, in which the monitor state is viewed as an abstract interpretation of sets of pairs of program runs.**

I. Introduction

Violations of information flow (IF) policies, such as using untrusted inputs without proper sanitization, can often be detected by means of data flow analysis. Such analysis can be done statically, though there is a tradeoff between soundness (catching all violations) and usability; sound analyses may produce many false positives due to conservative approximation. Dynamic analysis, or monitoring, costs runtime resources (e.g., for storage and propagation of labels), but it can be precise enough to avoid false positives. And it allows secure executions of imperfect programs. Unfortunately, data flow is the tip of the iceberg: an attacker wishing to avoid detection can exploit the flow of information via control (implicit flow), as well as various timing and other channels.

Information flow control (IFC) systems are designed to thwart implicit flows and, in some cases, other channels. In contrast with data flow analysis, IFC systems can provably enforce noninterference (NI) properties that are both mathematically precise and intuitively meaningful in connection with practical security requirements. Recent work demonstrates the possibility of dynamic IFC [1], [2], in particular for fine-grained policies that allow a program within a single thread and address space to manipulate information at multiple security levels. Unfortunately, there is a gap between the NI properties for which enforcement mechanisms are known and the security requirements which arise in practice.

Many policies both identify sensitive information and allow its *downgrading* under designated conditions. Untrusted input can be *endorsed*, i.e., trusted for certain purposes once it has been sanitized (e.g., used for SQL queries after being cleansed of embedded SQL fragments). Confidential information can be *declassified*, i.e., released for certain purposes once it has been suitably redacted, or anonymized, or is no longer timely. To provide a general way to express such IF policies, researchers have turned to forms of relational logic [3], [4]. Such logics describe pairs of program runs, as usual in notions of NI. In addition to being able to describe indistinguishability relations between the two runs, such logics include ordinary assertions that refer to the program state. The program state often includes what is relevant for policy, e.g., whether a timer has expired or user credentials have been checked successfully. It is also possible to add instrumentation to facilitate formulation of policy; such "ghost code" is common practice in software verification and has been suggested as a way to augment type-based IF policy [5].

We seek strong IFC for practical policies that involve downgrading. There are many challenges, such as how to obtain precise policy specifications and how to automate IFC for large software components. The specific problem that motivates this paper is how to design a runtime monitor to track policies expressed using relational logic. In a nutshell, our first contribution is a new relational logic with small-step semantics, catering for intermediate assumptions and assertions to express policies concerning inputs and outputs. Our second contribution is to show how monitored execution can be viewed as abstracting over the set of alternate executions. Our third contribution is deriving the design of a monitor, along with its correctness proof, by refining the semantic description into an implementable form.

A. Policies in relational logic

One of the motivations for use of relational logic instead of other policy notations is that program logics can provide expressive and precise reasoning about data structures. For example, sensitive data can appear as subsegments of string buffers, or as nodes in linked data structures involving sharing or aliasing. Logics provide ways to designate the mutable locations to which fine-grained policies apply [6], [4]. However, in order to focus on the main topics of this paper, we use a minimal imperative language.

An IFC system assumes that the designated secret variables/channels cannot be directly read and the designated trusted ones cannot be directly written by the adversary.

One technique for designating sensitive information is by labels used as types, or attached to types. For example, we could declare trusted_out: trusted, untrusted_out: untrusted, and hostname: untrusted. This is intended to express the integrity policy that trusted outputs are not influenced by untrusted inputs. Influence may be formalized in terms of pairs of executions: two runs that *agree on* (i.e., have the same values for) the trusted inputs should agree on the resulting trusted outputs. In the example below, varying the value of hostname should not affect the value of trusted_out. Confidentiality policy is interpreted similarly: two runs that agree on public inputs should agree on the final value of public outputs, independent of secret inputs.

We now turn to examples of conditional downgrading policies which are not so easily expressed as types or labels, inspired by Common Weakness Enumeration (CWE) [7].

The first example, inspired by CWE-15, is an endorsement policy: the input hostname should affect the actual name of the host (represented by the trusted_out channel) only if the password provided matches the root password. The policy is satisfied by this program:

```
input guess from public_in
input hostname from untrusted_in
input password from secret_in
if guess = password then output hostname to trusted_out;
else output "Access␣denied" to public_out;
```

In relational logic, variables are designated "low" by the ***agreement*** operator $\mathbb{A}$ in pre- and post-conditions (written $\#$ and $\ltimes$ in [8], [6]). For example, the contract

```
requires 𝔸 x    ensures 𝔸 y
```

expresses that any pair of runs that agree on the initial value of x should agree on the final value of y. We typically distinguish output variables from input variables; in that case the precondition expresses agreement on the low inputs and the postcondition expresses agreement on the low outputs. A practical syntax will provide defaults and syntax sugar, but in this paper we work directly with relational logic. In these terms we formalize the integrity policy as follows.

```
requires (𝔹 (public_in = secret_in) ⇒𝔸 untrusted_in)
         ∧ 𝔸 (public_in = secret_in)
ensures 𝔸 trusted_out
```

The *both* operator, $\mathbb{B}$, is used to say that a state predicate holds in both runs. The precondition says that if public_in = secret_in is true in both runs then they agree on untrusted_in. The postcondition says the runs agree on trusted_out. The precondition $\mathbb{A}$(public_in = secret_in) restricts attentions to pairs of runs that both do or both don't satisfy the equality of the guess and the password. A key point is that the implication in the precondition captures the policy in a way that directly refers to the program state. On the other hand, the policy specification is distinct from the executable code (which in practice might be idiomatic C).

The ability to express agreement for expressions is a key feature of relational logic. Specifications in the logic describe allowed dependency, which can capture both integrity and confidentiality policies. Next we consider a confidentiality policy in connection with the following undesirable code, inspired by CWE-204.

```
input user from public_in;
input guess from public_in;
input password from private_in;
if user != "root" then msg := "Incorrect␣username";
else if guess = password then msg := "Login␣successful";
    else msg := "Incorrect␣authentication␣credentials";
output msg to public_out;
```

We need to allow disclosing whether both the user name is found and the password matched, or not. But we should not allow disclosing partial matches —e.g., the user name was found, but the password didn't match— as it would lead to unnecessary disclosure. The following contract captures this requirement.

```
requires 𝔸 (user = "root" ∧ guess = password)
ensures 𝔸 public_out
```

It can be read as saying that by observing public output, the adversary is allowed to learn the value of the expression (user = "root"∧guess = password). Consider two runs that agree on falsity but for different reasons: in one run the user isn't "root", in the other run the password is wrong. The specification requires agreement on the output, which does not happen with the code above.

Consider a variation of the policy where we instead require $\mathbb{A}$(user = "root") ∧ $\mathbb{A}$(guess = password). It says that by observing the public output the adversary is allowed to depend on ("learn") the value of user = "root" and also to depend on password = masterPassword. This policy is satisfied by the vulnerable program presented earlier. Writing good policies can be difficult but is beyond the scope of this paper.

So far we have used pre/post contracts to express policies, which keeps policy and code separate. However, contracts are not always practical. Consider C programs that perform memory allocations via malloc. We would like to write a conditional endorsement policy that guards against CWE-789 by restricting the allocation size coming from an untrusted source. There is no single contract for malloc that is suited to all its uses, and it is hard to give an end-to-end contract for a large program using it. Instead our specification uses an assertion just before the call:

```
assert 𝔸 size; pt := malloc(size);
```

Elsewhere in the code, following where the tainted value of size is obtained, we use

```
assume 𝔸 (size < 1000) ∧ (𝔹 (size < 1000) ⇒𝔸 size)
```

In general, an assumption following an input plays the role of a requires clause, and an assertion preceding an output plays the role of an ensures. Contracts amount to an initial

assumption and a final assertion, so we do not formalize contracts as such.

As another example, recent cross-site scripting vulnerabilities in several Apache modules [9] can be prevented by requiring all untrusted inputs to be sanitized before sending to the client. In one such policy, we use

assume $\mathbb{A}$ html_escape(servername)

following the input of servername. Much later in the code, where the server name —supposed to have been sanitized— is be written to an output buffer, we use **assert** $\mathbb{A}$ buffer.

B. Monitoring

Several recent works present monitor designs for idealized languages, in which labels designate sensitive values and facilitate tracking of information. In many settings, including low-level LLVM code that is a focus of our project, it is important for labeling to be *flow-sensitive*; a memory location may hold secrets at one moment and public information at another. To avoid the cost and conservativity of static analysis, while allowing some flow sensitivity, purely dynamic monitoring is attractive [10]. There are tradeoffs between what can be achieved by purely dynamic monitoring versus *hybrid monitors* that rely on static analysis for branches not taken [1]. More related work is discussed in Sec. VI. Few of these works on monitoring address any form of downgrading. The most closely related work is by Askarov and Sabelfeld [11] who provide a flow-insensitive monitor that caters to declassification policies expressed by $declass$ statements as featured in Jif [12]. We seek to monitor rich policies and idiomatic code, where the downgrading may be performed by code different from the policy expression. For example, declassify h0+h1 by t:=h0; pubout:=t+h1.

The works cited above all provide correctness proofs with respect to precisely defined security properties. One challenge is to formulate a security property such that downgrading policies are interpreted in accord with intended requirements [13]. That is not the focus of this paper. Rather, we pursue the ideal that a correctness proof should provide a compelling explanation for the monitor design, so that theory could guide the design of practical monitors for production use. To date it is not easy to discern the commonalities and general principles among published proofs, or even the security properties.

That said, there are some basic features found in published security properties and correctness proofs, not only for monitors but also for static analyses. Security is described in terms of two executions, which are assumed to agree on low input events and must be shown to agree on low outputs. This is proved in terms of a more fine-grained property in which segments of a computation are classified as low or high contexts in accord with information in branch conditions. Low segments of the two computations are aligned in lockstep, whereas high segments may differ between the two executions but an agreement invariant is maintained (cf. "unwinding conditions"). While such correspondence is implicit in proofs based on bisimulation and the like, it is made explicit and called *alignment* by Kovács et al [14] who present a static analysis based on abstract interpretation of pairs of executions. A key point, as they discuss, is that existence of a good alignment suffices to prove NI, whereas refuting NI would require reasoning about all possible alignments. The choice of alignment is manifest in syntax in the works on static verification by self-composition [15], [16].

C. Overview and contributions

A monitor has to work on a single run, which we call the ***major trace*** following [17]. The monitor's instrumentation somehow tracks what could happen in alternate runs, called ***minor traces***. The main idea explored in this paper is that at each step the monitor state is an abstract representation of all possible minor traces consistent with the major trace. This representation must be such that it can be maintained as the computation proceeds. In this view, a monitor can allow a step of execution provided it can ensure that the chosen alignment(s) for every minor computation can be extended. When a monitoring rule requires raising a security error, it is because for some alternate run the chosen alignment cannot be extended in a way that maintains the agreement relation. This may be because the policy is indeed violated —in which case no alignment exists with the desired property— or because the chosen alignment is not one that can be used to establish NI for these runs. The latter case is a mechanism failure. Some mechanism failures are inevitable for a sound monitor. However, we aim for effective runtime reasoning about policy, e.g., leveraging runtime checking for ordinary assertions.

Section II lays the groundwork to explore our new view of monitoring: a simple programming language, and relational formulas for use in assumptions and assertions. Sec. III introduces a novel small-step semantics for relational specifications. To this end we introduce a novel classification of pairs of traces: some conform to the policy, some are irrelevant (because they diverge or violate assumptions), and some are failures because they violate assertions or cannot be aligned. A program is secure with respect to its specification if it admits no assertion failures. Roughly, this generalizes knowledge-based security properties like [18], [11].

In Sec. V we describe the job of a monitor as maintaining a *tracking set* for the major trace, i.e., an abstraction of the set of all possible minor traces, together with their classification. If there are any failures, the monitor should announce that the trace is unsafe.

For a monitor to track all minor traces it must be able to track an arbitrary one. In Sec. IV we define what a tracking should look like, for the various classifications. We show how the usual monitor apparatus —labels on

locations, plus a stack of security levels— describes the tracking of a minor trace, and we show how it can be updated as the major trace progresses. This provides a rational reconstruction of extant monitors. The monitor's transition relation can be defined in terms of the monitor state and underlying program configuration, independent of the minor trace. Hence the tracking set, of which the monitor state is an abstraction, can be maintained pointwise. Monitor transitions play the role of "transfer function" in the theory of abstract interpretation [19].

Because our policies are expressed in terms of assertions and assumptions, the usual monitor apparatus is not enough. We add one more component to the monitor state: a set of formulas known to hold for aligned pairs of configurations. By proving that this invariant can be maintained as the major trace grows (Thm. 7 in Sec. IV-B), we effectively derive a monitor that makes use of assumptions about inputs in checking assertions about outputs.

We find it remarkable that by describing monitoring as a kind of abstract interpretation, the actions of the monitor can be determined hand in hand with the explanation why it works correctly. We leave it to the reader to make the comparison with correctness proofs found in related works (often lengthy unpublished appendices).[1] But this paper is just a first step. Sec. VII discusses questions about our approach that need further investigation. A key goal is to formalize an abstraction function according to the theory of abstract interpretation, and to "turn the crank" for a formal derivation of the monitor. Work of this kind is included under related work, Sec. VI. We challenge ourselves and others to see whether the approach helps in the design and implementation of monitors for practical programming languages and deployment environments.

II. Background

Section II-A defines a small-step operational semantics for simple imperative programs, the basis for our formal development. Sec. II-B defines a relational assertion language to be used for policies expressed in the assertion and assumption commands. The following Sec. III formalizes the security property in these terms.

A. Programming Language

Simple imperative language with annotation commands

$e ::= n \mid x \mid e \oplus e \quad (n \in \mathbb{Z}, x \in \mathit{Vars})$	integer expr.
$e ::= \mathsf{tt} \mid \mathsf{ff} \mid x \mid e = e \mid e \leq e \mid e \wedge e \mid \neg e$	boolean expr.
$c ::= \mathbf{skip} \mid x := e \mid c\,;c \mid \mathbf{while}\ e\ \mathbf{do}\ c$	commands
$\quad \mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c$	
$\quad \mid \mathbf{assert}^{\ell}\ \Phi \mid \mathbf{assume}^{\ell}\ \Phi\ (\ \ell \in \mathit{Tags})$	annotations

Variables are assumed to have a fixed type, either integer or boolean. Expressions are required to be type-correct. We eschew formalizing the typing of expressions and commands. Relational formulas Φ are defined in the sequel. Tags ℓ serve to make occurrences of annotation commands unique and may be elided when there is no ambiguity. For brevity we use the term ***annotation*** rather than "annotation command".

The small-step semantics is very standard. ***Configurations*** take the form $\langle c,\ \sigma\rangle$ where c is a command and σ a state. Let $code\langle c,\ \sigma\rangle$ be c and $state\langle c,\ \sigma\rangle$ be σ. A ***state*** is a mapping from variables to their values, which are integers or booleans. Henceforth, σ and τ range over states. We write $\sigma(e)$ for the value of expression e in state σ, and assume that expressions are defined in all states. We write $[\sigma|x : v]$ for the updated state that maps x to v. In concrete examples we write states like $[x : 2, y : \mathsf{tt}]$.

Transition rules for commands

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

$$\langle x := e,\ \sigma\rangle \mapsto \langle \mathsf{skip},\ [\sigma|x : \sigma(e)]\rangle$$

$$\langle \mathsf{skip}; c,\ \sigma\rangle \mapsto \langle c,\ \sigma\rangle \qquad \frac{\langle c,\ \sigma\rangle \mapsto \langle c_1,\ \tau\rangle}{\langle c; d,\ \sigma\rangle \mapsto \langle c_1; d,\ \tau\rangle}$$

$$\langle \mathsf{while}\ e\ \mathsf{do}\ c,\ \sigma\rangle \mapsto \langle \mathsf{if}\ e\ \mathsf{then}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip},\ \sigma\rangle$$

$$\frac{\sigma(e) = v}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_{\mathsf{tt}}\ \mathsf{else}\ c_{\mathsf{ff}},\ \sigma\rangle \mapsto \langle c_v,\ \sigma\rangle}$$

Programs are deterministic and annotations have no effect.

Trace notations

A ***trace*** is a finite non-empty sequence of configurations that is consecutive under the transition relation. Let T, U, V range over traces. We write sequences by catenation[2] and refer to the length as $|T|$. We also treat sequences as functions from an initial segment of the naturals. So T_0 is the first configuration and $dom(T)$ is the set of indices $0, \ldots, |T|-1$. Write $last(T)$ for $T_{|T|-1}$. Write $T\rfloor i$ for the first i elements of T, i.e., the prefix up to but not including the element T_i. Write $T' \geq T$ to say T is a prefix of T'. An example trace: $\langle \mathbf{assume}\Phi; \mathbf{assert}\Psi, \sigma\rangle\langle \mathbf{skip}; \mathbf{assert}\Psi, \sigma\rangle\langle \mathbf{assert}\Psi, \sigma\rangle\langle \mathbf{skip}, \sigma\rangle$.

We assume any considered program c is well formed in the sense of being type-correct, a property that is preserved by the transition relation. We also assume that in any initial configuration $\langle c,\ \sigma\rangle$ considered, (a) no annotation tag ℓ occurs more than once in c, and (b) c has the form $b; \mathbf{skip}$ for some b. In a more concrete semantics, instead of tags we would use program points; tagged syntax caters for use of

[1]This is not meant as a criticism of those works, which focus on other important questions such as what are good security properties or how can monitoring be done efficiently.

[2]For example, we can write $T\langle c,\ \sigma\rangle$ for a trace with $\langle c,\ \sigma\rangle$ as its last configuration. Also, $\langle c,\ \sigma\rangle$ is a trace, of length 1.

structural operational semantics. The transition rule for **while** duplicates the loop body, so tags are not necessarily unique in non-initial configurations. Item (b) loses no generality and helps reduce the number of cases to be considered in some proofs. For the same reason, we have chosen to make if/then the only branching transition. We gloss over (b) in examples.

In Sec. IV and V we restrict attention to programs of the form **assume**$\Phi; c;$ **skip**.

Configurations of the form $\langle$**skip**$, \sigma\rangle$ are terminated. Any other reachable configuration has the form $\langle c, \sigma\rangle$ where c can be factored as a sequence $c \equiv b; d$ and b is not a sequence. Then b is called $redex(c)$, as it is the sub-command re-written by the next transition, and d is called $remainder(c)$. Let $redex\langle c, \sigma\rangle$ be $redex(c)$. Note that $redex($**skip**$; c)$ is **skip**, whereas $redex($**skip**$)$ is undefined.

Lemma 1. For any trace T and any $0 < i < |T|$, if $redex(T_i)$ is an annotation command then $redex(T_{i-1})$ is not an annotation command.

B. Relational formulas

Relational assertions amount to assertions on a pair of states, so any first order language can be adapted to a language of relational formulas. However, we are concerned with relations that connect the two states by equalities — agreements.

Relational formulas Φ for annotations

φ	$::=$	$e \mid \varphi \vee \varphi \mid \neg\varphi \mid \forall x.\varphi$	unary formula
Φ	$::=$	$\mathbb{A}e \mid \mathbb{B}\varphi \mid \mathbb{B}\varphi \Rightarrow \mathbb{A}e$	basic relational formula
Φ	$::=$	$\Phi \wedge \Phi$	conjunction of basic formulas

Unary formulas φ are ordinary first order formulas; the atomic formulas are boolean expressions of the programming language, and formulas are evaluated in a single state. ***Relational*** formulas, ranged over by the letters Φ and Ψ, are evaluated in a pair of states. Relational formulas feature conditional agreements, $\mathbb{B}\varphi \Rightarrow \mathbb{A}e$, that require both states to agree on the value of e if φ holds in both states. Reasoning about conditional agreements is facilitated by square relations of the form $\mathbb{B}\varphi$. An unconditional agreement $\mathbb{A}e$ is equivalent to $\mathbb{B}\mathsf{tt} \Rightarrow \mathbb{A}e$.

We use standard semantics for first order formulas: $\sigma \models \varphi$ means φ is true in state σ and $\not\models$ indicates that it is false. Note that $\sigma \models e$ iff $\sigma(e) = \mathsf{tt}$, for boolean expression e.

Semantics of relational formulas $\quad \sigma|\tau \models \Phi$

$\sigma|\tau \models \mathbb{A}e$	iff	$\sigma(e) = \tau(e)$
$\sigma|\tau \models \mathbb{B}\varphi$	iff	$\sigma \models \varphi$ and $\tau \models \varphi$
$\sigma|\tau \models \mathbb{B}\varphi \Rightarrow \mathbb{A}e$	iff	$\sigma|\tau \models \mathbb{B}\varphi$ implies $\sigma|\tau \models \mathbb{A}e$
$\sigma|\tau \models \Phi \wedge \Psi$	iff	$\sigma|\tau \models \Phi$ and $\sigma|\tau \models \Psi$

Notation: $\langle c, \sigma\rangle|\langle c', \sigma'\rangle \models \Psi$ means $\sigma|\sigma' \models \Psi$

Free variables (FV) are defined in the standard way.

Lemma 2. Suppose $\sigma(x) = \sigma'(x)$ and $\tau(x) = \tau'(x)$ for all $x \in FV(\Psi)$. Then $\sigma|\tau \models \Psi$ iff $\sigma'|\tau' \models \Psi$.

To define the monitor semantics, we work with ***extended relational formulas*** of the form $\Phi \Rightarrow \Psi$ where Ψ may be $\mathbb{A}\varphi$ or a basic relational formula; and also simply $\mathbb{A}\varphi$. For extended relational formulas, the semantics is defined as the notation suggests: $\sigma|\tau \models \mathbb{A}\varphi$ means that $\sigma \models \varphi$ iff $\tau \models \varphi$. And $\sigma|\tau \models \Phi \Rightarrow \Psi$ means that either $\sigma|\tau \not\models \Phi$ or $\sigma|\tau \models \Psi$. We write $\models \Phi \Rightarrow \Psi$ to indicate that the implication is ***valid***, i.e., $\sigma|\tau \models \Phi \Rightarrow \Psi$ for all σ, τ.

III. Small-step relational logic

To enable use of intermediate assertions and assumptions to express downgrading policies, we introduce a novel small-step notion of relational correctness. Its formulation caters for the abstract interpretation described in subsequent sections of the paper, and for termination-insensitive security. We define what it means for a trace to be safe. A program is secure, i.e., correct with respect to its specification, iff all its traces are safe.

Alignment, proper alignment, coverage

For traces T, U, an ***alignment*** from T to U is a relation $\alpha \subseteq dom(T) \times dom(U)$ 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*** from T to U is an alignment α such that for all i, j with $i\alpha j$, if either $redex(T_i)$ or $redex(U_j)$ is an annotation then $redex(T_i) = redex(U_j)$.

For α to ***cover the major trace*** (resp. the minor trace) means that $dom(\alpha) = dom(T)$ (resp. $rng(\alpha) = dom(U)$).

Aligned annotations have the same identifying tag, not just the same formula. Also, in a proper alignment that covers both traces, if one trace has executed an annotation command then the other has too. To be precise:

Lemma 3. Let α be a proper alignment that covers both T and U. Suppose $i\alpha j$ and $redex(T_i)$ is an annotation. Then $i < |T| - 1$ iff $j < |U| - 1$.

In general, proper alignments are not unique. As an illustration, consider two traces of a program **assume**$\mathbb{A}$y; x:= y; **assert**$\mathbb{A}$x which differ only in starting states. Two (of several) possible proper alignments for the two traces are shown in Fig. 1, where commands **assume**Ψ and **assert**Ψ are abbreviated as $[\Psi]$ and $\{\Psi\}$ respectively.

The following notions classify the ways in which a minor trace U may relate to a major trace T. On first reading it may be helpful to think about a command of the form **assume**$\Phi; c;$ **assert**Ψ where c has no annotations.

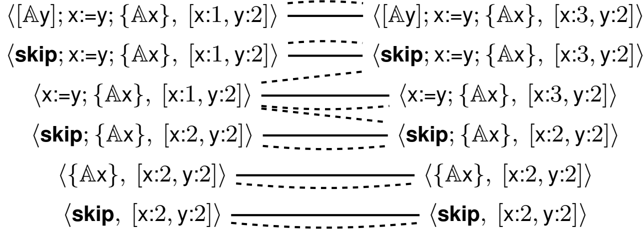

Figure 1. An example of two proper trace alignments (and conformances). Trace T goes downward on the left, trace U on the right. One alignment is shown as solid lines, another as dashed lines.

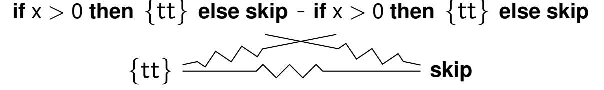

Figure 2. Alignment failure example. Zigzag lines show incorrect alignments pairs. Only the code is shown; on the left, x is initially 1, on the right 0. The initial configurations are aligned, but there is no proper alignment that covers the traces.

Classification of aligned trace pairs

An ***aligned trace pair*** is a triple (T, U, α) where α is a proper alignment from T to U.

A ***conformance*** is an aligned trace pair (T, U, α) where α covers both T and U, and for all i, j, if $i\alpha j$, $i < |T| - 1$, and $redex(T_i)$ asserts or assumes Φ then $T_i|U_j \models \Phi$.

An ***assumption failure*** is (T, U, α) where there are i, j, ℓ, Φ such that $i < |T|$, $j < |U|$, $(i-1)\alpha(j-1)$, $redex(T_{i-1})$ is $\textsf{assume}^{\ell}\Phi$, $T_i|U_j \not\models \Phi$, and $(T{\lfloor}i, U{\lfloor}j, \beta)$ is a conformance, for $\beta = \{(k, l) \mid k\alpha l \wedge k < i \wedge l < j\}$.

An ***assertion failure*** is the same as an assumption failure, except that $redex(T_{i-1})$ is $\textsf{assert}^{\ell}\Phi$.

An ***alignment failure*** is (T, U, α) where there is i, $i < |T|$, such that $redex(T_i)$ is an annotation command, $(T{\lfloor}i, U, \alpha)$ is a conformance, and there is $U' > U$ such that either $last(U')$ is terminated or $redex(last(U'))$ is an annotation different from $redex(T_i)$, and no proper prefix of U' extends U and matches $redex(T_i)$.

A ***divergence failure*** is (T, U, α) where there is i, $i < |T|$, such that $redex(T_i)$ is an annotation command, $(T{\lfloor}i, U, \alpha)$ is a conformance, and there are infinitely many $U' \geq U$, none of which has $redex(last(U')) = redex(T_i)$.

The idea is that a computation T is secure if there are no alternate traces resulting in assertion or alignment failure. Assumption failures generalize preconditions, ruling out some traces as not being of interest. We consider divergence failures benign, which embodies the decision to focus on termination-insensitive security properties.

Conformance allows the traces to end with an aligned annotation that has not been executed and whose formula may not hold. Note that $(i-1)\alpha(j-1)$ implies $0 < i$ and $0 < j$.

For assumption failure, the redex of the last configuration in $T{\lfloor}i$ (and in $U{\lfloor}j$) is the assumption. This fits with the definition of conformance, which does not constrain an unexecuted assumption. Note also that T and U may continue past i, j. Finally, instead of $T_i|U_j \not\models \Phi$ we could as well write $T_{i-1}|U_{j-i} \not\models \Phi$ as the transition for an assumption does not change the state.

Observe that, similar to proper alignments in general, conformances are not unique. Both alignments presented in figure 1 are indeed conformances because they are proper and they happen to satisfy the assertions and assumptions at trace points $(0, 0)$ and $(4, 4)$.

To illustrate assertion failures, we will slightly modify our running example by removing x:=y. In this example the alignment is unique. If both traces hadn't made the last step the alignment would have been considered a conformance.

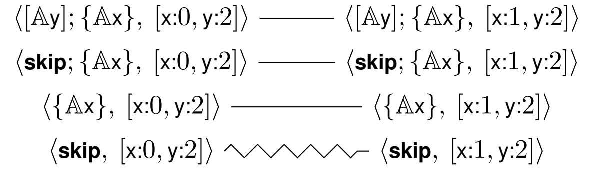

Fig. 2 illustrates alignment failure. Alignment failure for an assumption would indicate implicit flow into a downgrading, which is considered bad policy. Our security property enforces one aspect of robust declassification [20], [11], which allows a low observer to learn from the declassified value but not from the fact that a declassification has occurred.

Safe trace

Let c be a command and T a trace of c. The trace T is ***safe*** iff for every state σ there is a trace U from $\langle c, \sigma \rangle$ and alignment α from T to U such that (T, U, α) is a conformance, an assumption failure, or a divergence failure.

Recall that typical security properties quantify over states σ that are low-equivalent to $state(T_0)$, whereas here we quantify over all σ. That works because the policy is intended to be expressed by an initial assumption that serves to define low equivalence. Low-equivalence in final states can be designated by a final assertion. Suppose the program has an initial assumption and a final assertion, both conjunctions of agreements on variables. If there are no other annotations, safety is exactly the standard notion of termination-insensitive NI.

In the definition of safety, σ may be instantiated by $state(T_0)$. Say T has a ***unary assertion failure*** if it reaches $\langle d, \tau \rangle$ where $redex(d)$ is $\textsf{assert}\,\mathbb{B}\varphi \wedge \ldots$ and $\tau \not\models \varphi$. If $(T, T, id_{dom(T)})$ is a conformance then T is free of unary

assertion failures. Thus, insofar as the program has assertions of the form $\mathbb{B}\varphi$, our notion of security includes absence of unary assertion failures.

To conclude this section, the following Theorem says that the trace classification is exhaustive. Its proof foreshadows subsequent proofs which lead us to monitoring, and it motivates details in the classification of aligned trace pairs.

Lemma 4. For any T with $|T| > 1$, if $(T\lfloor(|T|-1), U, \alpha)$ is an assumption-, assertion-, alignment-, or divergence-failure then so is (T, U, α).

Theorem 5. For any command c and trace T of c, and any state σ, there is $U \geq \langle c,\ \sigma\rangle$ and α such that (T, U, α) is either a conformance or one of the four forms of failure.

Proof: By induction on T. For the base case, $|T|$ is 1. Let U be the singleton trace $\langle c,\ \sigma\rangle$ and α be $\{(0,0)\}$. This is a conformance.

For the induction step, suppose T is $V\langle d, \tau\rangle$. By induction hypothesis there are U, α such that (V, U, α) is a failure or conformance. In case of failure, $(V\langle d, \tau\rangle,\ U,\ \alpha)$ is the same kind of failure, by Lemma 4. It remains to consider the case that (V, U, α) is a conformance. We proceed by cases on whether $redex(d)$ is an annotation command.

In case $redex(d)$ is an annotation, we extend U by steps without active assumptions or assertions until reaching a configuration where the redex is the same as $redex(d)$, if possible. More precisely, let U' be the shortest trace such that $U' \geq U$, $redex(last(U')) = redex(d)$, and none of the redexes of $U'_{|U|}, \ldots, U'_{|U'|-2}$ are annotation commands. If no such U' exists because a non-matching annotation command is reached or the computation terminates, then $(V\langle d,\ \tau\rangle,\ U,\ \alpha)$ is an alignment failure. If no such U' exists because the computation diverges without reaching any annotation command, $(V\langle d,\ \tau\rangle,\ U,\ \alpha)$ is a divergence failure. If U' does exist, let α' be $\alpha \cup \{(|V|, |U'|-1)\} \cup \{(|V|-1, j) \mid |U| \leq j < |U'|-1\}$. Observe that α' is a proper alignment from T to U'. (To see this, note that $redex(last(V))$ is not an assertion or assumption because it is followed by d; see Lemma 1.) Finally, $(V\langle d, \tau\rangle,\ U', \alpha')$ is a conformance.

In case $redex(d)$ is not an annotation, we go by cases on $redex(last(V))$.

- If $redex(last(V))$ is $\mathsf{assume}^{\ell}\Phi$, then by proper alignment we have $redex(last(U)) = redex(last(V))$ and $(|V|-1)\alpha(|U|-1)$. Let U' be $U\langle d',\ \tau'\rangle$ where $last(U) \mapsto \langle d',\ \tau'\rangle$. Let $\alpha' = \alpha \cup \{(|V|, |U|)\}$. If $\tau|\tau' \models \Phi$ then $(V\langle d,\ \tau\rangle,\ U', \alpha')$ is a conformance; otherwise it is an assumption failure.
- In case is $\mathsf{assert}^{\ell}\Phi$, we proceed as in the preceding case, but instead of an assumption failure or conformance we get assertion failure or conformance.
- In case $redex(last(V))$ is not an annotation, let α' be $\alpha \cup \{(|V|, |U|-1)\}$. Because (V, U, α) is a conformance, $last(U)$ is not an annotation, so α' is a proper alignment from $V\langle d,\ \tau\rangle$ to U', and $(V\langle d,\ \tau\rangle,\ U', \alpha')$ is a conformance.

Owing to Lemma 1, these are the only cases. ■

IV. How to track an aligned minor trace

For a given major trace, a static analysis or runtime monitor will attempt to find alignments that witness the existence of minor trace U and α in the definition of safety; and it will attempt to check assertions, given the preceding assumptions. These attempts may fail. When a monitor raises a security exception, it means the monitor is no longer able to ensure that execution is secure. We use the term ***mechanism failure*** for these.

This section considers what a monitor might do with respect to a single minor trace. To figure out how the monitor should work, we give constructive proofs of two results that strengthen Thm. 5. The second proof essentially defines a monitor, in the sense that the construction does not refer to the minor trace. This means that the monitor is effectively tracking all minor traces, as we spell out in Sec. V.

In the rest of the paper we require all programs to take the form $\mathsf{assume}\ \Phi; c$. The assumption is needed in order to determine sensible initialization for the monitor state.

A. Strong conformances

We introduce some ingredients one would expect in a hybrid monitor. The first step adds augmentation, similar to that in the monitors of [21], [1], that allows us to define the particular alignments of interest. The second step, in Sec. IV-B, makes better use of annotation. We also use a rudimentary static analysis: let $targets(c)$ be the set of variables that occur on the left side of assignments in c.

Augmented configuration

An ***augmented*** configuration is $\langle c,\ \sigma,\ \lambda, \pi\rangle$ where the variable levels are a map $\lambda : Vars \to \{\mathbf{lo}, \mathbf{hi}\}$ and the program counter level π is either the token $\mathbf{lo}$ or a tuple $(\mathbf{hi}, stk)$ where stk is a non-empty list of pairs (b, c) of commands. Some notations:

$\lambda(e)$	$=$	$\sqcup(map\,\lambda\,(vars(e)))$	level of expr.
$\mathbb{A}\lambda$	$=$	$\wedge\{\mathbb{A}x \mid \lambda(x) = \mathbf{lo}\}$	derived agrmnt.
$\lfloor\langle c,\ \sigma,\ \lambda, \pi\rangle\rfloor$	$=$	$\langle c,\ \sigma\rangle$	erasure
$pc\langle c,\ \sigma,\ \lambda, \pi\rangle$	$=$	π	get pc levels
$levels\langle c,\ \sigma,\ \lambda, \pi\rangle$	$=$	λ	get var. levels
$lift(\lambda)(X)$	$=$	λ'	
		where $\lambda'(x) = \mathbf{hi}$, if $x \in X$	
		$\lambda'(x) = \lambda(x)$, otherw.	

An ***augmented trace*** is a sequence of augmented configurations whose erasure is a trace (i.e., consecutive under transitions $\mapsto$).

The idea is that in a trace segment where the context is high, i.e., $\pi = (\mathbf{hi}, (b, c) : rest)$ the command c is the "low

continuation" that represents the control flow join point and the command b is the branch not taken in the major trace (which provides for static analysis). It is for c to always exist that we need the trailing **skip** in initial configurations. A stack is needed to handle programs like this:

if h $>$ 0 **then** (**if** h $>$ 1 **then** m := 0 **else skip**) **else** n := 0

Consider trace T from initial state with $h = 1$, versus minor trace from a state with $h = 2$. The minor trace follows the same top level path, but not the same inner path.

The notions of alignment, conformance, and failure are carried over to aligned pairs (T, U, α) where T is a trace of augmented configurations and U a trace of ordinary configurations.

Strong conformance

A ***strong conformance*** is a conformance (T, U, α) such that T is an augmented trace and the following hold for all i, j with $i\alpha j$.

(a) if $i > 0$ and $j > 0$ then $T_i|U_j \models \mathbb{A}(levels(T_i))$,
(b) if $pc(T_i) = \mathbf{lo}$ then $code(U_j) = code(T_i)$,
(c) if $pc(T_i) = (\mathbf{hi}, (b, c) : rest)$ then there is k such that $k < i$ and $code(T_k)$ has the form $F; c$ where $F =$ **if** e **then** b_{tt} **else** b_{ff} (for some $e, b_{\mathsf{tt}}, b_{\mathsf{ff}}$), b is $b_{\neg v}$ and $v = state(T_k)(e)$, i.e., b is the branch not taken. Moreover, either
(i) $code(T_i) = d; c$ and $code(U_j) = F; c$, for some d with $b_v \mapsto^* d$, or
(ii) $code(T_i) = c$ and $code(U_j) = d; c$, for some d with $F \mapsto^* d$.

and *mutatis mutandis* for the rest of the stack.

For commands b, b' we write $b \mapsto^* b'$ to abbreviate $\exists\sigma, \sigma'.\langle b,\ \sigma\rangle \mapsto^* \langle b',\ \sigma'\rangle$.

The proof of Thm. 5 suggests that a monitor might track a minor trace by incrementally extending the alignment. We next strengthen Thm. 5 to strong conformances. The decision to incrementally extend a single alignment, rather than search among possible alignments, is obvious from an algorithmic point of view but it comes at the price of approximation. A chosen alignment may not be extendable yet another may exist. For example, in two runs of the program in Fig. 2, if both follow the true branch there will be a proper alignment, but if the monitor treats the conditional as a high context then that alignment will not be found. In practical terms, the situations where this may arise can be seen as poorly chosen policies.

The obvious thing to do is introduce a notion of ***mechanism failure***: an aligned trace pair (T, U, α) where $(T\lfloor i, U, \alpha)$ is a conformance, for some i. A property like Thm. 5 but allowing mechanism failure is essentially vacuous. What one would hope for is constraints on the circumstances under which we declare mechanism failure. In this paper we do not formalize such constraints (nor have we found that done in prior work). Instead we focus on deriving a sensible monitor. We appeal to mechanism failure only where it seems unavoidable.

Safety now means absence of mechanism failure, alignment failure, and assertion failure. A difficulty is how to ignore divergence failures. As defined in Sec. III, such a failure occurs at the point when the major trace reaches an annotation. In reasoning about strong conformance, however, we need to consider the possibility of divergence of the minor run at the join points of high branches.

Eventual divergence failure, quiescent conformance

(T, U, α) is an ***eventual divergence failure*** if it is a (strong) conformance and there is some $T' \geq T$ such that (T', U, α) is a divergence failure.

(T, U, α) is a ***quiescent conformance*** if it is a (strong) conformance and there does not exist any $T' \geq T$ such that $redex(last(T'))$ is an annotation.

A divergence failure is a special case of eventual divergence failure. For termination-insensitive security it makes sense to ignore eventual divergence failures. The point of quiescent conformance is that further execution of the major trace cannot lead to assertion or alignment failure and so need not be considered a violation of safety.

Theorem 6 [Strong conformance]. Consider any ordinary trace V of a command c_0 from state σ_0, and any state τ. Augmentation can be added to obtain a trace T, with $\lfloor T \rfloor = V$, such that there is a trace U of c_0 from τ, and an alignment α, and either (T, U, α) is a strong conformance or it is one of the failures, including mechanism failure, eventual divergence, or quiescent conformance.

Proof: By induction on V. There are two base cases, for V of length 1 and 2. Suppose c_0 is $\mathsf{assume}\Psi; c_1$. (Recall that every initial program begins with an assumption.) Choose λ_0 such that $\models \Psi \Rightarrow \mathbb{A}\lambda_0$. (In the proof of Thm. 7 we discuss how this can be implemented.) In case $|V| = 1$, let T be $\langle c_0,\ \sigma_0,\ \lambda_0, \mathbf{lo}\rangle$. Let $U := \langle c_0,\ \tau\rangle$ and $\alpha := \{(0, 0)\}$. This is a strong conformance. In the other base case, $|V| = 2$, let T be $\langle c_0,\ \sigma_0,\ \lambda_0, \mathbf{lo}\rangle\langle \mathsf{skip}; c_1,\ \sigma_0,\ \lambda_0, \mathbf{lo}\rangle$. Let $U := \langle c_0,\ \tau\rangle\langle \mathsf{skip}; c_1,\ \tau\rangle$. Let $\alpha := \{(0, 0), (1, 1)\}$. If $\sigma_0|\tau \models \Psi$ then this is a strong conformance; otherwise it is an assumption failure.

We sketch the induction step, for the interesting case where conformance holds so far. Suppose (T, U, α) is a strong conformance. Suppose $last(T)$ is $\langle c,\ \sigma,\ \lambda, \pi\rangle$ and $\langle c,\ \sigma\rangle \mapsto \langle c',\ \sigma'\rangle$. We must find $\lambda', \pi', U', \alpha'$ such that $(T\langle c',\ \sigma',\ \lambda', \pi'\rangle, U', \alpha')$ is either a strong conformance or one of the failure forms. We go by cases on π.

In case $\pi = \mathbf{lo}$ we go by cases on $redex(c)$:

$\underline{x := e}$: Let $\pi' := \mathbf{lo}$, $\lambda' := [\lambda | x : \lambda(e)]$, $U' := U\langle d,\ \tau\rangle$ where $last(U) \mapsto \langle d,\ \tau\rangle$, and $\alpha' := \alpha \cup \{(|T|, |U|)\}$. The strong conformance property holds for the added pair $(|T|, |U|)$, because $redex(c) = redex(last(U))$ and if the level $\lambda(e)$ is $\mathbf{lo}$ then $\sigma'(x) = \tau(x)$ as required.

$\underline{\mathsf{while}\ e\ \mathsf{do}\ d}$: Let $\lambda', \pi' := \lambda, \mathbf{lo}$ and, as in the preceding case, $\alpha' := \alpha \cup \{(|T|, |U|)\}$ and $U' := U\langle d,\ \tau\rangle$ where $last(U) \mapsto \langle d,\ \tau\rangle$. This yields strong conformance; the **while** redex is not a branching one and the step has no effect on state.

$\underline{\mathsf{skip}}$: λ', π', U', and α' are the same as for **while**.

$\underline{\mathsf{if}\ e\ \mathsf{then}\ c_{\mathsf{tt}}\ \mathsf{else}\ c_{\mathsf{ff}}}$: If $\lambda(vars(e)) = \mathbf{lo}$ then $\lambda', \pi', U', \alpha'$ are the same as in the **while** case. Otherwise, let $v := \sigma(e)$, $\lambda' := \lambda$, $\pi' := (\mathbf{hi}, (c_{\neg v}, d) : [])$ where $d = remainder(c)$, i.e., d is what follows the if/then command. Either way, we have strong conformance.

$\underline{\mathsf{assert}\Psi}$: If $last(T)|last(U) \models \Psi$ then we get a strong conformance, same as **while**. Otherwise, $(T\langle c',\ \sigma',\ \lambda, \pi\rangle, U\langle c',\ \tau\rangle, \alpha \cup \{(|T|, |U|)\})$ is an assertion failure, where $last(U) \mapsto \langle c',\ \tau\rangle$. (We know the configuration will be c'.)

$\underline{\mathsf{assume}\Psi}$: If $last(T)|last(U) \models \Psi$ then same as **while**. Otherwise, assumption failure as in preceding case.

In case $\pi = (\mathbf{hi}, (b, d) : rest)$ we again go by cases on $redex(c)$:

$\underline{x := e}$: Let $\lambda' := [\lambda | x : \mathbf{hi}]$, $\pi' := \pi$, $U' := U$, $\alpha' := \alpha \cup \{(|T|, |U| - 1)\}$, which yields a strong conformance.

$\underline{\mathsf{while}\ e\ \mathsf{do}\ d'}$: Let $\lambda' := \lambda$, $\pi' := \pi$, $U' := U$, and $\alpha' := \alpha \cup \{(|T|, |U| - 1)\}$, which yields a strong conformance.

$\underline{\mathsf{skip}}$: If $c' \neq d$, this step is continuing in a high segment of the major trace, so let $\lambda', \pi', U' := \lambda, \pi, U$, and $\alpha' := \alpha \cup \{(|T|, |U| - 1)\}$.

Otherwise $c' = d$, i.e., this step reached the end of a high segment in the major trace. Let $\lambda' := lift(\lambda, targets(b))$. Also $\pi' := (\mathbf{hi}, rest)$ if $rest \neq []$, and $\pi' := \mathbf{lo}$ if $rest = []$.

Furthermore, by definition of strong conformance there's $\mathsf{if}\ e\ \mathsf{then}\ b_{\mathsf{tt}}\ \mathsf{else}\ b_{\mathsf{ff}}$ such that b is b_{tt} or b_{ff} and $code(last(U))$ is $(\mathsf{if}\ e\ \mathsf{then}\ b_{\mathsf{tt}}\ \mathsf{else}\ b_{\mathsf{ff}}); d$. Let V be the trace with initial configuration $last(U)$, up to and including the step from $\mathsf{skip}; d$ to d. In case V exists, let $U' := U \mathbin{+\!\!+} V$ and let $\alpha' := \alpha \cup \{(|T|, j) \mid |U| \leq j < |U'|\}$. This forms a strong conformance.

It may be that V does not exist, because the computation from $last(U)$ diverges. In that case, we augment the last major step by $\lambda', \pi', U', \alpha' := \lambda, \pi, U, \alpha$. If further execution of the major trace can reach an annotation then we have an eventual divergence failure. If not, we have a quiescent conformance.

$\underline{\mathsf{if}\ e\ \mathsf{then}\ b_{\mathsf{tt}}\ \mathsf{else}\ b_{\mathsf{ff}}}$: As in the case for **while**, let $\lambda', U', \alpha' := \lambda, U, \alpha \cup \{(|T|, |U| - 1)\}$. However, π' needs to record the alternative branch. As in the case for if/then in a low context, let $v := \sigma(e)$ and $\pi' := (\mathbf{hi}, (b_{\neg v}, d') : (b, c) : rest)$ where the remainder d' is defined by $c = (\mathsf{if}\ e\ \mathsf{then}\ b_{\mathsf{tt}}\ \mathsf{else}\ b_{\mathsf{ff}}); d$.

$\underline{\mathsf{assert}\Psi}$: $(T\langle c',\ \sigma',\ \lambda, \pi\rangle, U, \alpha)$ is deemed to be a mechanism failure (though it may well be leading to an alignment failure).

$\underline{\mathsf{assume}\Psi}$: Same as **assert**.

■

B. Tracking known formulas

The construction in the preceding proof is interesting because determining new values λ', π' for the variable levels and program counter, and determining whether there is some kind of failure, is based largely on the current configuration of the major trace. It does refer to the minor trace to evaluate asserted and assumed formulas. But the minor trace is a semantic artifact with which to reason about information flow —a monitor will have no access to the minor trace. Our next step eliminates that, and improves the treatment of λ' and π' by taking advantage of assumptions and assertions.

Consider as an example the fragment

assume $\mathbb{A}$ (h0 + h1); x := h1 + h0; **assert** $\mathbb{A}$ x

in a low context, where the initial λ has $\lambda(h0) = \mathbf{hi}$ and $\lambda(h1) = \mathbf{hi}$. According to the proof of Thm 6, the final levels λ'' will have $\lambda''(x) = \mathbf{hi}$, but if there is no assumption failure then the two traces will agree on x. Update of the levels needs to take into account the assumption. There is a similar issue for if/then: In a low context, determination of whether an if/then initiates a high context is based on whether the two traces agree on the guard expression e. Rather than simply checking whether $\lambda(e) = \mathbf{lo}$, it would be better to take prior assumptions into account, for example, when e is h0+h1>0 following the assumption $\mathbb{A}$(h0 + h1).

Fully augmented configuration, full conformance

A ***fully augmented configuration*** takes the form $\langle c,\ \sigma,\ \lambda, \pi, \Delta\rangle$ where λ and π are as in an augmented configuration and Δ is a set of basic relational formulas.

A ***full conformance*** is a strong conformance (T, U, α) where T is a trace of fully augmented configurations and for all $i > 0, j > 0$, if $i\alpha j$ then $T_i | U_j \models \Delta$.

The set Δ is interpreted conjunctively, so we may treat Δ as a formula, and we write $\sigma|\tau \models \Delta$ to mean $\sigma|\tau \models \Phi$ for every Φ in Δ.

One could dispense with λ, by including in Δ the formula $\mathbb{A}x$, for each x with $\lambda(x) = \mathbf{lo}$. We retain λ, both for clarity and because in an implementation the two sorts of information may best have different representations.

For an assignment $x := e$ in a low context, $\lambda'(x)$ can be **lo** if $\models \mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}e$. (Recall that $\mathbb{A}\lambda$ abbreviates a conjunction of variable agreements $\mathbb{A}x, \mathbb{A}y, \ldots, \mathbb{A}z$ where $x, y, \ldots z$ are the low variables according to λ.) Note that the condition $\models \mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}e$ refers to validity of the formula, rather than its truth in the current states —information about the current states is provided by λ and Δ. This condition also serves for determining whether there is agreement on the guard expression of an if/then.

The next question is how to check annotations, i.e., how to determine whether $\langle c, \sigma, \lambda, \pi, \Delta\rangle|\langle c', \sigma'\rangle \models \Psi$ without looking at σ'. In addition to appealing to the full conformance condition $\sigma|\sigma' \models \mathbb{A}\lambda \wedge \Delta$, it is possible to check unary assertions in the major trace. There are interesting connections between unary and relational assertions. For example, $\mathbb{B}(x = 0)$ implies $\mathbb{A}x$, and more generally $\mathbb{B}\varphi$ for any φ that implies a specific value for x. Also $\sigma \models \varphi$ and $\sigma|\tau \models \mathbb{A}\varphi$ imply $\tau \models \varphi$. These considerations suggest that it could be useful to know the strongest invariant at each program point; approximations can be obtained by static analysis. An appropriate interface to the monitor would be to provide this information in the form of designated assumptions, as we discuss in Sec. V. Assertions are checked, not trusted by our monitor, but it remembers the results from successful checks.

The monitor can rely on functions that approximate checks of truth and validity, specified as follows.

Specification of truth and validity checkers

$eval(\sigma, \varphi) \in \{\mathsf{tt}, \mathsf{unknown}\}$
$eval(\sigma, \varphi) = \mathsf{tt}$ implies $\sigma \models \varphi$
$chkVal(\Psi \Rightarrow \Phi) \in \{\mathsf{tt}, \mathsf{unknown}\}$, for basic formulas Φ
$chkVal(\Psi \Rightarrow \Phi) = \mathsf{tt}$ implies $\models \Psi \Rightarrow \Phi$

For $eval$, a particularly simple implementation is the program semantics: let $eval(\sigma, e) = \mathsf{tt}$ if $\sigma(e) = \mathsf{tt}$, for boolean expression e, and $eval(\sigma, \varphi) = \mathsf{unknown}$ otherwise.

For $chkVal$, we will be particularly interested in checks where the antecedent includes the variable agreements given by $\mathbb{A}\lambda$. We sketch a simple implementation that caters for this.

$chkVal(\Psi \Rightarrow \mathbb{A}e)$ $= \mathsf{tt}$, if $\mathbb{A}x$ is in Ψ, for every $x \in FV(e)$
$= \mathsf{unknown}$, otherwise
$chkVal(\Psi \Rightarrow \mathbb{A}\varphi)$ $= \mathsf{tt}$, if $\mathbb{A}x$ is in Ψ, for every $x \in FV(\varphi)$
$= \mathsf{unknown}$, otherwise
$chkVal(\Psi \Rightarrow \mathbb{B}\varphi)$ $= \mathsf{tt}$, if Ψ contains $\mathbb{B}\varphi$
$= \mathsf{unknown}$, otherwise

The truth and validity checkers are used by the following function. It approximates the checking of $\sigma|\tau \models \Phi$, with σ explicitly given but τ known only to be related according to λ and Δ.

Assertion checker $check(\sigma, \lambda, \Delta)(\Phi) \in \{\mathsf{tt}, \mathsf{ff}, \mathsf{unknown}\}$

$check(\sigma, \lambda, \Delta)(\mathbb{B}\varphi)$
$= \mathsf{ff}$, if $eval(\sigma, \neg\varphi) = \mathsf{tt}$
or $chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{B}\neg\varphi) = \mathsf{tt}$
$= \mathsf{tt}$, if $chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{B}\varphi) = \mathsf{tt}$
$= \mathsf{tt}$, if $chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}\varphi) = \mathsf{tt}$ and $eval(\sigma, \varphi) = \mathsf{tt}$
$= \mathsf{unknown}$, otherwise
$check(\sigma, \lambda, \Delta)(\mathbb{A}e)$
$= \mathsf{tt}$, if $chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}e) = \mathsf{tt}$
$= \mathsf{unknown}$, otherwise
$check(\sigma, \lambda, \Delta)(\mathbb{B}\varphi \Rightarrow \mathbb{A}e)$
$= \mathsf{tt}$, if $eval(\sigma, \neg\varphi) = \mathsf{tt}$
or $chkVal(\mathbb{A}\lambda \wedge \Delta \wedge \mathbb{B}\varphi \Rightarrow \mathbb{A}e) = \mathsf{tt}$
$= \mathsf{unknown}$, otherwise
$check(\sigma, \lambda, \Delta)(\Phi_0 \wedge \Phi_1)$
$= \mathsf{tt}$, if $check(\sigma, \lambda, \Delta)(\Phi_i) = \mathsf{tt}$ for $i = 0$ and $i = 1$
$= \mathsf{ff}$, if $check(\sigma, \lambda, \Delta)(\Phi_i) = \mathsf{ff}$ for $i = 0$ or $i = 1$
$= \mathsf{unknown}$, otherwise

More sophisticated implementations are possible, including ones that take advantage of guardedness. This refers to formulas like $\mathbb{A}e \wedge (\mathbb{B}e \Rightarrow \mathbb{A}e')$, as in examples in Sec. I-A.

Theorem 7. For any command c and initial states σ and τ, any trace of c from σ can be augmented to a trace T such that there is an ordinary trace U of c from τ, and an alignment α, such that (T, U, α) is either a full conformance or one of the forms of failure, including eventual divergence and quiescent conformance.

Proof: By induction on the given trace. The base cases are the same as in the proof of Thm. 6, with the addition that Δ_0, in the first configuration of the major trace, is $\emptyset$. The second configuration's Δ is Ψ where the initial assumption of c is Ψ.

For the induction step to longer major traces, suppose (T, U, α) is a full conformance. Suppose $last(T)$ is $\langle c,\ \sigma,\ \lambda, \pi, \Delta\rangle$ and $\langle c,\ \sigma\rangle \mapsto \langle c',\ \sigma'\rangle$. We must find $\lambda', \pi', \Delta', U', \alpha'$ such that $(T\langle c',\ \sigma',\ \lambda', \pi', \Delta'\rangle, U', \alpha')$ is either a full conformance or one of the failure forms.

By cases on $redex(c)$ and cases on whether π is **lo**.
In case $\pi = \mathbf{lo}$ we go by cases on $redex(c)$:

<u>$x := e$</u>: As in the proof of Thm. 6, we let $\pi' := \mathbf{lo}$, $U' := U\langle d,\ \tau\rangle$ where $last(U) \mapsto \langle d,\ \tau\rangle$, and $\alpha' := \alpha \cup \{(|T|, |U|)\}$. Let $\Delta' = \{\Phi \mid \Phi \in \Delta \wedge x \notin FV(\Phi)\}$. Owing to Lemma 2 we have $\sigma|\tau \models \Delta'$. Let $\lambda' := [\lambda|x : l]$ where l is **lo** if either $\lambda(e) = \mathbf{lo}$ or $chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}e) = \mathsf{tt}$. (With stronger assumptions on $chkVal$, there would be no need for checking $\lambda(e) = \mathbf{lo}$.)[3]

[3]For low assignments, the Δ' chosen in the proof can be seen as a crude approximation of the strongest postcondition for the assignment acting on each of the basic relational formulas in Δ. Better approximations are within easy reach but would distract from our main focus. A more nuanced treatment is also possible for the join point in a high segment.

c	π	π'	λ'	Δ'
x:=e; d	$\mathbf{lo}$	π	$[\lambda \mid x : l]$ where $l = \begin{cases} \mathbf{lo} \text{ if } \lambda(e) = \mathbf{lo} \vee \\ \quad chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}e) \\ \mathbf{hi} \text{ otherwise} \end{cases}$	$\{\Phi \mid \Phi \in \Delta \wedge x \notin FV(\Phi)\}$
x:=e; d	$(\mathbf{hi}, _)$	π	$[\lambda \mid x : \mathbf{hi}]$	$\{\Phi \mid \Phi \in \Delta \wedge x \notin FV(\Phi)\}$
while e do p; d	$_$	π	λ	Δ
skip; d	$\mathbf{lo}$	π	λ	Δ
skip; d	$(\mathbf{hi}, (b, d) : [])$	$\mathbf{lo}$	$lift(\lambda, targets(b))$	$\{\Phi \mid \Phi \in \Delta \wedge targets(b) \cap FV(\Phi) = \emptyset\}$
skip; d	$(\mathbf{hi}, (b, d) : r)$	$(\mathbf{hi}, r)$	$lift(\lambda, targets(b))$	$\{\Phi \mid \Phi \in \Delta \wedge targets(b) \cap FV(\Phi) = \emptyset\}$
skip; d	$(\mathbf{hi}, (b, c') : _)$	π	λ	Δ
if e then b_{tt} else b_{ff}; d	$\mathbf{lo}$	$\begin{cases} \mathbf{lo} \text{ if } \lambda(vars(e)) = \mathbf{lo} \vee \\ \quad chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}e) \\ (\mathbf{hi}, (b_{\neg\sigma(e)}, d) : []) \text{ otherwise} \end{cases}$	λ	Δ
if e then b_{tt} else b_{ff}; d	$(\mathbf{hi}, st)$	$(\mathbf{hi}, (b_{\neg\sigma(e)}, d) : st)$	λ	Δ
assert Ψ; d	$\mathbf{lo}$	failure, if $check(\Psi, \sigma, \lambda, \Delta) \neq \mathsf{tt}$; otherwise, see next row:		
		π	λ	Δ, Ψ
assert Ψ; d	$(\mathbf{hi}, _)$	failure		
assume Ψ; d	$\mathbf{lo}$	π	λ	Δ, Ψ
assume Ψ; d	$(\mathbf{hi}, _)$	failure		

Table I
MONITORING RULES: DEFINE λ', π' AND Δ' IN $\langle c,\ \sigma,\ \lambda, \pi, \Delta\rangle \mapsto \langle c',\ \sigma',\ \lambda', \pi', \Delta'\rangle$

while e do d: Let $\Delta' := \Delta$ and the rest be as in the proof of Thm. 6. In subsequent cases we just give the additions and differences from that proof.

skip: Let $\Delta' := \Delta$.

if e then c_{tt} else c_{ff}: If $\lambda(vars(e)) = \mathbf{lo}$ or $chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{A}e)$ then same as the case of **while** above; otherwise, same as in Thm. 6 for entering a high branch. Either way, let $\Delta' := \Delta$.

assertΨ: If $check(\sigma, \lambda, \Delta)(\Psi) = \mathsf{ff}$ we have an assertion failure. If $check(\sigma, \lambda, \Delta)(\Psi) = \mathsf{tt}$ we have a conformance, in which case the formulas of Ψ can be recorded, i.e., $\Delta' := \Delta, \Psi$. Otherwise we declare a mechanism failure.

assumeΨ: Let $\Delta' := \Delta, \Psi$. Observe that $(T\langle c', \sigma', \lambda, \pi, \Delta'\rangle, U', \alpha')$ is either a full conformance —if $last(T)|last(U) \models \Psi$— or an assumption failure, so we are done. Without recourse to checking whether $last(T)|last(U) \models \Psi$, it is not possible to determine which it is.

In case $\pi = (\mathbf{hi}, (b, d) : rest)$ we again go by cases on $redex(c)$:

$x := e$: As in the proof of Thm. 6, let $\lambda' := [\lambda|x : \mathbf{hi}]$, $\pi' := \pi$, $U' := U$, $\alpha' := \alpha \cup \{(|T|, |U| - 1)\}$. Let $\Delta' = \{\Phi \mid \Phi \in \Delta \wedge x \notin FV(\Phi)\}$. As in the case of low assignment, this maintains the full conformance property.

while e do d: Let $\Delta' := \Delta$.

skip: If $c' \neq d$, this step is continuing in a high segment of the major trace. We extend the construction in Thm. 6 by $\Delta' := \Delta$. Otherwise, $c' = d$, i.e., this step reached the end of a high segment in the major trace. Again we extend the construction in Thm. 6, in this case with $\Delta' := \{\Phi \mid \Phi \in \Delta \wedge targets(b) \cap FV(\Phi) = \emptyset\}$. Either way, the rest goes as in Thm. 6.

if e then c_{tt} else c_{ff}: As in Thm. 6, adding $\Delta' := \Delta$.

assertΨ: As in Thm. 6.

assumeΨ: As in Thm. 6.

■

The constructive part of the proof can be summarized in the form of monitoring rules, given in Table I. The table defines transition relations for monitored configurations, providing new values of the variable levels, control context, and known formulas, given the current configuration. It should be read as follows: Pick the first row where the values of c and π in the current monitor configuration match the patterns ("_" matches everything) in the respective columns. E.g., if current full configuration has c that looks like (**skip**; d) and π is $\mathbf{lo}$, then the values of π', λ' and Δ' in the next configuration are same as in the current. (The new values of c' and σ' are defined by the language semantics.) The table indicates conditions under which there is failure. In Sec. V we discuss what the monitor can do upon detecting failure. By contrast with the proof of Thm. 7, we do not distinguish between assertion failure and mechanism failure in the checking of assertions.

The initial value of π is $\mathbf{lo}$. The initial values of λ and Δ are obtained from the first assumption in the program. Specifically, we stipulate that any program should look like: **assume**Ψ; c; **skip** for some Ψ and c. Then, the initial value of Δ is going to be Ψ (treated as a set of basic formulas) and the initial value of λ such that $\models \Psi \Rightarrow \mathbb{A}\lambda$. A simple example of a monitored execution is presented in Fig. 3.

step	c	σ	λ	π	Δ
0	**assume**$\mathbb{A}$y; x:=y; **assert**$\mathbb{A}$x; **skip**	[x:1, y:2]	[x:**hi**, y:**lo**]	**lo**	$\mathbb{A}y$
1	**skip**; x:=y; **assert**$\mathbb{A}$x; **skip**	[x:1, y:2]	[x:**hi**, y:**lo**]	**lo**	$\mathbb{A}y$
2	x:=y; **assert**$\mathbb{A}$x; **skip**	[x:1, y:2]	[x:**hi**, y:**lo**]	**lo**	$\mathbb{A}y$
3	**skip**; **assert**$\mathbb{A}$x; **skip**	[x:2, y:2]	[x:**lo**, y:**lo**]	**lo**	$\mathbb{A}y$
4	**assert**$\mathbb{A}$x; **skip**	[x:2, y:2]	[x:**lo**, y:**lo**]	**lo**	$\mathbb{A}y$
5	**skip**; **skip**	[x:2, y:2]	[x:**lo**, y:**lo**]	**lo**	$\mathbb{A}y$
6	**skip**	[x:2, y:2]	[x:**lo**, y:**lo**]	**lo**	$\mathbb{A}y$

Figure 3. Example monitor trace

V. Monitoring as abstract interpretation

The preceding section considered how a single minor run can be tracked by a major run equipped with suitable instrumentation. However, the use and update of instrumentation is independent of the particular minor run. This is the key to monitoring: the monitor computes an abstraction of the set of possible minor traces.

It is convenient to use the following tokens to represent the classification of trace alignments: $ignore$, $fail$, ok.

Tracking sets

A ***tagged minor trace for*** T is a quadruple (T, U, α, tag) such that $code(T) = code(U)$ and (a) if tag is $ignore$, (T, U, α) is an assumption failure, eventual divergence, or quiescent conformance; (b) if tag is $fail$, (T, U, α) is as assertion failure, alignment failure, or designated as a mechanism failure; and (c) if tag is ok then (T, U, α) is a full conformance.

A ***tracking set for*** T is a set X of tagged minor traces for T with the following properties.

(Completeness) For all σ, there is some (T, U, α, tag) in X such that $state(U_0) = \sigma$.

(Irredundancy) For all (T, U, α, tag) and (T, U', α', tag') in X, if $state(U_0) = state(U'_0)$ then $U = U'$, $\alpha = \alpha'$, and $tag = tag'$.

We include $state(T_0)$ among the states σ: As discussed earlier, the monitor is checking ordinary unary assertion failures too.

Here is an immediate consequence of the definitions.

Theorem 8. If X is a tracking set for T and X does not contain the tag $fail$ then T is safe (i.e., $\lfloor T \rfloor$ is safe).

To determine the safety of a trace T it suffices to maintain a tracking set. This can be done by pointwise updates, owing to the construction in Theorems 6 and 7.

Theorem 9 [Concretion of monitored execution]. Suppose V is an ordinary trace. Then there is a fully augmented trace T, with $\lfloor T \rfloor = V$, and a tracking set for T.

Proof: By induction on V. In the base case, V is a singleton trace $\langle c,\ \sigma \rangle$. Let T be $\langle c,\ \sigma,\ \lambda, \pi, \Delta \rangle$ where the initial monitor state λ, π, Δ is given in the proof of Thm. 7. Let $\alpha := \{(0,0)\}$. Let X be the set $\{(T, \langle c,\ \tau \rangle, \alpha, ok) \mid \tau \in States\}$. This is a tracking set.

For the induction step consider a trace $V\langle c,\ \sigma \rangle$. By induction, let T and X be the fully augmented trace and tracking set for V. Extend T and each tagged minor trace in X according to the construction in the proof of Thm. 7, choosing the appropriate tag in each case. Note that extension of T is determined entirely by the current configuration, independent of any minor trace (Table I). In some cases it is unknown, "from the monitor's point of view", which kind of failure occurs, but it is always determined which tag applies. ■

Monitoring: Having determined how the monitor tracks minor traces, what remains is to decide how it should handle failure. One possibility is to maintain a boolean flag, initially true and set false if failure reached. The major trace is secure so long as the flag is true. For strong security, the computation should be halted if failure is reached. Simply logging the failure may be appropriate for software testing. Failures may hint at faulty policy, or the need for additional assumptions to facilitate reasoning by the monitor.

The monitor is evidently transparent, in the sense that it does not alter the program behavior except for whatever action is taken upon failure. However, a safe trace may be considered as a failure, due to choice of alignment as well as incompleteness of assertion checking.

What about multi-level policies? In the relational logic approach to information flow policy, such policies are expressed by using multiple relational specifications. One may also want multiple policies because there are multiple requirements. In our setting, one could choose for annotation labels pairs (p, n) where p is a policy identifier and n provides uniqueness. A program may have many annotations, all with unique labels but several may share a given policy name p. For each p there can be a monitor based on the annotations for p, ignoring the other annotations. The cartesian product of these monitors enforces all the policies. There are obvious optimizations of the product monitor, e.g., the product of several λ functions can be represented by a single one that maps to a product lattice instead of $\{\mathbf{lo}, \mathbf{hi}\}$.

Static and dynamic reasoning about annotations: Here is an example of how the monitor does some reasoning.

```
assume 𝔸 (guess = password)
        ∧ (𝔹 (guess = password) ⇒𝔸 untrusted)
if guess = password then assert 𝔹 (guess = password);
                              trusted := untrusted
assert 𝔸 trusted
```

When the branch is taken, the intermediate assertion is in a low context (owing to the initial assumption). It checks successfully, because guess = password holds in the current state and the monitor still knows the agreement $\mathbb{A}$ (guess = password). For the assignment, the monitor determines the label on untrusted can be set to **lo**, because from $\mathbb{B}$ (guess = password) and the initial assumption it knows $\mathbb{A}$ untrusted.

Here is an example where hybrid monitoring can benefit from static assertion checking.

```
assume 𝔸 (h0+h1) ∧ 𝔸 (h2+h3)
x02 := h0 + h2; x13 := h1 + h3; y := x02; y := y + x13;
assert 𝔹 (y = h0 + h1 + h2 + h3);
out := y;
assert 𝔸 out
```

Semantically, both assertions are valid. In fact the intermediate assertion can be established by a rudimentary unary program verifier, and for any valid unary assertion φ the assertion $\mathbb{B}\varphi$ is valid in relational logic. The monitor presented here cannot establish the first assertion: Although it can test y = h0 + h1 + h2 + h3 in the current state, and it can deduce $\mathbb{A}$ (h0 + h1 + h2 + h3) from the initial assumption, it cannot deduce $\mathbb{A}$ y.

The point of the example is that, although prior "hybrid" monitors used static analysis to determine potentially updated state, another useful form of static analysis is ordinary assertion checking. In this case, that justifies *assuming* $\mathbb{B}$(y = h0 + h1 + h2 + h3), which then serves to inform the monitor about the current value of y so that it can deduce the agreement and thereby confirm the final assertion. One can also use assumptions in this way for statically checked relational assertions. Of course such assumptions should be distinguished from those that are part of the policy specification.

Another policy issue is that, for some end-to-end security properties, it is important to disallow mutation of variables that occur in downgrading policy assumptions. For the monitor to track whether a given location has been assigned or still has its initial value, one can add "old-expressions" to the assertion language, for use in assertions like x = old(x). Old-expressions have been implemented in several runtime assertion checkers [22].

VI. Related work

Relational logic for specification of information flow has been introduced by Benton [23] and Amtoft and Banerjee [8] for simple imperative programs. Hunt and Sands [24] provide a flow-sensitive type system that is equivalent. Amtoft et al [6] extend the logic to object-based programs by means of a region-based heap abstraction, so agreements can be expressed for heap locations. Their idea inspired development of region logic [25], a relational version of which is work in progress. Amtoft et al [26] implement relational contracts for SparkAda procedures, which can express conditional agreements for array segments, including generation of verification conditions using relational loop invariants. To specify policies involving multiple levels they introduce indexed agreements which compactly encode multiple requires/ensures contracts. Nanevski et al [4] design and implement Relational Hoare Type Theory, providing machine-checked static verification of relational policies for program features including dynamic allocation and deallocation. Their policies are less expressive than ours insofar as they only provide pre/post contracts. On the other hand, their relations are expressed using full higher order logic; in particular, this provides for an unusual notion of declassification based on opacity of abstract predicates.

Banerjee et al [18] propose a mix of relational logic and type checking to express policies with declassification, extending the end-to-end property of [27]. Another extension that addresses the "what" and "where" dimensions of declassification is given in [11]. We conjecture that by choosing a suitable class of annotations in our relational logic, we can obtain their security properties as consequence of our semantics in Sec. III.

The idea of making alignments an explicit structure, with a minor run constructed incrementally appeared in [18] where it was used for soundness proof of their static analysis. Kovács et al. [14] formulate and implement abstract interpretation of trace pairs; we adopt their term "alignments". Their algorithm checks pre/post properties. They construct alignments from trace pairs by automatic syntactic matching. In contrast, works like [16], [15] use static analysis and transformation to construct a syntactic representation that determines the alignment for all pairs.

The extra components in augmented configurations (our λ and π) have been used in existing information flow monitors [1], [21], [28], [2], [17], [10], [29], [11]. Trace alignments are not explicit in prior work on monitoring, except for [2] which has hinted at their existence, calling them "trace pairs": "corresponding" elements of two traces where one of the elements might not exist, and the pairing is according to program counter values explicit in traces.

The idea of a "hybrid" monitor that uses static analysis to account for indirect implicit flows when control flow merges goes back to [2] and [21]. However, as shown in [1], the former did not need to be hybrid after all, due to being flow-insensitive. Moore and Chong [30] adapt the monitor of [1] to programs that dynamically allocate state, showing what sorts of heap abstraction can support hybrid monitoring. Beringer [17] delves into the design of flow-sensitive monitors, dissecting the components of hybrid monitors dealing with direct and aspects of indirect flow, studying the guarantees given by variations on existing

enforcement mechanisms. He coined the terms “minor” and “major” which we use.

Most of the previous work on monitoring was concerned with plain NI policies. Askarov and Sabelfeld’s [11] stands out as the only one, to the best of our knowledge, to address declassification policies. The monitor carries around a set E declassified expressions, which at first glance resembles our Δ. But the monitor’s behavior is not affected by E; it only serves as ghost instrumentation with which to define the security property, an end-to-end knowledge based property like those of [18], [11], [31]. Because policy expression is conflated with the release action, the monitor can just execute the declass as an assignment (and add the expression to E). Agreement on the values of released expressions is in the initial memory rather than the current one. Owing to the flexibility of relational logic, their security property can be expressed in our setting by judicious choice of policy including specification of which variables can be modified. Birgisson et al [32] observe that many integrity policies simply require invariance of a predicate or value. They augment the monitor of [11] to check an ordinary assertion upon termination (effectively using old-expressions).

Our treatment of assumptions and assertions resembles the security property used by Dupressoir et al [33] for (unary) reasoning about protocol implementations. Flow-locks [5] extend information-flow typing with state-dependent conditions. Austin et al [34] describe dynamic IFC for imperative programs by translation to a lambda calculus. Secure multi-execution [35], [36] and faceted execution [37] are being explored as alternatives to monitoring by label-tracking.

VII. Discussion

The connection made here between information flow monitoring and relational program logic is just a beginning, which points to many topics for further investigation. In ongoing work we are developing an inlined monitor for the LLVM intermediate representation that implements the design described in this paper. This involves a richer assertion language to deal with arrays and buffered streams. We are also exploring how to make a hybrid monitor tunable in terms of how much it relies on static analysis, including verification of assertions, prior to execution. We can also tune the performance/completeness tradeoff for runtime reasoning ($eval$ and $chkVal$). A useful feature is detection of inconsistent assumptions.[4]

Although abstract interpretation is the leading idea of the paper, our formalization does not make explicit use the general theory of abstract interpretation. Prior work shows how abstract transfer functions can be derived by calculation from concrete semantics together with an abstraction function [19], [38]. Unlike these works, however, we are dealing with a hyperproperty so the concrete domain and abstraction function are somewhat complicated. We leave this to future work, aiming to build on the work of Kovács et al [14].

There has been other work on abstract interpretation for information flow and even declassification, but using abstractions of single executions [39]. A general framework could facilitate investigation of alternate monitoring techniques such as fully dynamic monitoring [10], [40].

One should also investigate relationships among different security properties and how they may or may not be expressible in our relational logic or a variation of it to cater for termination-sensitive security properties. Our development leans heavily on determinacy, so nondeterminacy is a particularly interesting challenge.

Acknowledgments: Thanks to Anindya Banerjee and the anonymous reviewers for helpful feedback. Chudnov and Naumann were partially supported in part by Department of Homeland Security, Science and Technology Directorate (contract 11027-202037-DS to HRL Laboratories). Naumann was partially supported by NSF award CNS-1228930.

[4]For example, some checking can be done by the monitor as described in this paper. If $\mathbb{B}\varphi$ is a basic formula among those being assumed, and φ is false in the current major state, or $chkVal(\mathbb{A}\lambda \wedge \Delta \Rightarrow \mathbb{B}\neg\varphi) = \mathtt{tt}$, the assumption is inconsistent.

References

[1] A. Russo and A. Sabelfeld, “Dynamic vs. static flow-sensitive security analysis,” in *IEEE Computer Security Foundations Symposium*, 2010.

[2] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar, “Provably correct runtime enforcement of non-interference properties,” in *Information and Communications Security*, ser. LNCS, vol. 4307, 2006.

[3] 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.

[4] A. Nanevski, A. Banerjee, and D. Garg, “Dependent type theory for verification of information flow and access control policies,” *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 2, 2013.

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

[6] T. Amtoft, S. Bandhakavi, and A. Banerjee, “A logic for information flow in object-oriented programs,” in *ACM Symposium on Principles of Programming Languages*, 2006.

[7] “The Common Weakness Enumeration (CWE) Initiative,” accessed February 2014. [Online]. Available: http://cwe.mitre.org/

[8] T. Amtoft and A. Banerjee, “Information flow analysis in logical form,” in *SAS*, 2004, pp. 100–115.

[9] “Vulnerability Summary for CVE-2012-3499.” [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=DCVE-2012-3499

[10] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *ACM Workshop on Programming Languages and Analysis for Security*, 2009.

[11] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," *IEEE Computer Security Foundations Symposium*, 2009.

[12] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. (2001-, July) Jif: Java + information flow, Software release. [Online]. Available: http://www.cs.cornell.edu/jif

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

[14] 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.

[15] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *Formal Methods*, ser. LNCS, vol. 6664, 2011.

[16] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Static Analysis Symposium*, ser. LNCS, vol. 3672, 2005.

[17] L. Beringer, "End-to-end multilevel hybrid information flow control," in *ACM Workshop on Programming Languages and Analysis for Security*, ser. LNCS, 2012, vol. 7705.

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

[19] P. Cousot and R. Cousot, *Basic Concepts of Abstract Interpretation*. Kluwer Academic Publishers, 2004.

[20] S. Zdancewic and A. Myers, "Robust declassification," in *IEEE Computer Security Foundations Symposium*, 2001.

[21] 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.

[22] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," in *Formal Methods for Components and Objects*, ser. LNCS, 2003, vol. 2852.

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

[24] S. Hunt and D. Sands, "On flow-sensitive security types," in *ACM Symposium on Principles of Programming Languages*, 2006.

[25] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Local reasoning for global invariants, part I: Region logic," *Journal of ACM*, vol. 60, no. 3, pp. 18:1–18:56, 2013.

[26] T. Amtoft, J. Hatcliff, and E. Rodríguez, "Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays," in *European Symposium on Programming*, ser. LNCS, vol. 6012, 2010.

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

[28] P. Shroff, S. F. Smith, and M. Thober, "Securing information flow via dynamic capture of dependencies," *J. Comput. Secur.*, vol. 16, no. 5, 2008.

[29] T. H. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *ACM Workshop on Programming Languages and Analysis for Security*, 2010.

[30] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *IEEE Computer Security Foundations Symposium*, 2011.

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

[32] A. Birgisson, A. Russo, and A. Sabelfeld, "Unifying facets of information integrity," in *Information Systems Security - 6th International Conference*, ser. LNCS, vol. 6503, 2010.

[33] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," *Journal of Computer Security*, 2014.

[34] T. H. Austin, C. Flanagan, and M. Abadi, "A functional view of imperative information flow," in *Asian Symposium on Programming Languages and Systems*, ser. LNCS, vol. 7705, 2012.

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

[36] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," in *IEEE Computer Security Foundations Symposium*, 2013.

[37] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *POPL*, 2012.

[38] J. Midtgaard and T. P. Jensen, "A calculational approach to control-flow analysis by abstract interpretation," in *Static Analysis Symposium*, ser. LNCS, vol. 5079, 2008.

[39] I. Mastroeni and A. Banerjee, "Modelling declassification policies using abstract domain completeness," *Mathematical Structures in Computer Science*, vol. 21, no. 6, 2011.

[40] J. Magazinius, A. Askarov, and A. Sabelfeld, "Decentralized delimited release," in *Asian Symposium on Programming Languages and Systems*, ser. LNCS, vol. 7078, 2011.