Van Delft, B., Hunt, S. & Sands, D. (2015). Very static enforcement of dynamic policies. Lecture Notes in Computer Science, 9036, pp. 32-52. doi: 10.1007/978-3-662-46666-7_3

Very Static Enforcement of Dynamic Policies

Bart van Delft[1], Sebastian Hunt[2], and David Sands[1]

[1] Chalmers University of Technology, Sweden
[2] City University London

Abstract. Security policies are naturally dynamic. Reflecting this, there has been a growing interest in studying information-flow properties which change during program execution, including concepts such as declassification, revocation, and role-change.
A static verification of a dynamic information flow policy, from a semantic perspective, should only need to concern itself with two things: 1) the dependencies between data in a program, and 2) whether those dependencies are consistent with the intended flow policies as they change over time. In this paper we provide a formal ground for this intuition. We present a straightforward extension to the principal flow-sensitive type system introduced by Hunt and Sands (POPL '06, ESOP '11) to infer both end-to-end dependencies and dependencies at intermediate points in a program. This allows typings to be applied to verification of both static and dynamic policies. Our extension preserves the principal type system's distinguishing feature, that type inference is independent of the policy to be enforced: a single, generic dependency analysis (typing) can be used to verify many different dynamic policies of a given program, thus achieving a clean separation between (1) and (2).
We also make contributions to the foundations of dynamic information flow. Arguably, the most compelling semantic definitions for dynamic security conditions in the literature are phrased in the so-called knowledge-based style. We contribute a new definition of knowledge-based progress insensitive security for dynamic policies. We show that the new definition avoids anomalies of previous definitions and enjoys a simple and useful characterisation as a two-run style property.

1 Introduction

Information flow policies are security policies which aim to provide end-to-end security guarantees of the form "information must not flow from this source to this destination". Early work on information flow concentrated on static, multi-level policies, organising the various data sources and sinks of a system into a fixed hierarchy. The policy determined by such a hierarchy (a partial order) is simply that information must not flow from a to b unless $a \sqsubseteq b$.

1.1 Dynamic policies

Since the pioneering work of Denning and Denning [DD77], a wide variety of information-flow policies and corresponding enforcement mechanisms have been proposed. Much of the recent work on information-flow properties goes beyond the static, multi-level security policies of earlier work, considering instead more sophisticated, dynamic forms of policy which permit different flows at different points during the excecution of

a program. Indeed, this shift of focus better reflects real-world requirements for security policies which are naturally dynamic.

For example, consider a request for sensitive employee information made to an employer by a regulatory authority. In order to satisfy this request it may be necessary to temporarily allow the sensitive information to flow to a specific user in the Human Resources department. In simplified form, the essence of this example is captured in Figure 1.

```
// x → a
out x on a;
// x ↛ a
out 2 on a;
```

Fig. 1

Here x contains the sensitive information, channel a represents the HR user, and the policy is expressed by the annotations $\mathtt{x} \rightarrow a$ (x *may* flow to a) and $\mathtt{x} \not\rightarrow a$ (x *must not* flow to a). It is intuitively clear that this program complies with the policy.

Consider two slightly more subtle examples, in each of which revocation of a permitted flow depends on run-time data:

```
1  /*Program A*/
2  // x, y → a
3  out x on a;
4  if (y > 0) {
5    out 1 on a;
6    // x ↛ a
7  }
8  out 2 on a;
9  out 3 on a;
```

```
1  /*Program B*/
2  // x → a
3  out x on a;
4  if (x > 0) {
5    out 1 on a;
6    // x ↛ a
7  }
8  out 2 on a;
9  out 3 on a;
```

In program A, the revocation of $\mathtt{x} \rightarrow a$ is controlled by the value of y, whereas in program B it is controlled by the value of x itself. Note that the policy for A explicitly allows $\mathtt{y} \rightarrow a$ so the conditional output (which reveals information about y) appears to be permissible. In program B the conditional output reveals information about x itself, but this happens *before* the revocation. So should program B be regarded as compliant? We argue that it should not, as follows. Consider "the third output" of program B as observed on channel a. Depending on the initial value of x, the observed value may be either 2 (line 8) or 3 (line 9). Thus this observation reveals information about x and, in the cases where revocation occurs, the observation happens *after* the revocation.

Unsurprisingly, increasing the sophistication of policies also increases the challenge of formulating good semantic definitions, which is to say, definitions which both match our intuitions about what the policies mean and can form the basis of formal reasoning about correctness.

At first sight it might seem that increasing semantic sophistication should also require increasingly intricate enforcement mechanisms. However, all such mechanisms must somehow solve the same two distinct problems:

1. Determine what data dependencies exist between the various data sources and sinks manipulated by the program.
2. Determine whether those dependencies are consistent with the flows permitted by the policy.

Ideally, the first of these problems would be solved independently of the second, since dependencies are a property of the code, not the policy. This would allow reuse at two levels: a) reuse of the same dependency analysis mechanisms and proof techniques for different *types* of policy; b) reuse of the dependency properties for a given program across verification of multiple *alternative* policies (whether of the same type or not).

In practice, enforcement mechanisms are typically not presented in a way which cleanly separates the two concerns. Not only does this hamper the reuse of analysis mechanisms and proof techniques, it also makes it harder to identify the *essential* differences between different approaches.

Central Contribution We take a well-understood dependency type system for a simple while-language, originally designed to support enforcement of static policies, and extend it in a straightforward way to a language with output channels (§ 5). We demonstrate the advantages of a clean separation between dependency analysis and policy enforcement, by establishing a generic soundness result (§ 6) for the type system which characterises the meaning of types as dependency properties. We then show how the dependency information derived by the type system can be used to verify compliance with dynamic policies. Note that this means that the core analysis for enforcement can be done even before the policy is known: we dub this *very static* enforcement. More significantly, it opens the way to reuse dependency analyses across verification of multiple types of information flow policy (for example, it might be possible to use the dependency analyses from advanced slicing tools such as JOANA [JOA] and Indus [Ind]).

Foundations of Dynamic Flow Policies Although it was not our original aim and focus, we also make some contributions of a more foundational nature, and our paper opens with these (§2–§4). The semantic definition of security which we use is based on work of Askarov and Chong [AC12], and we begin with their abstract formulation of dynamic policies (§2). In defining security for dynamic policies, they made a convincing case for using a family of attackers of various strengths, following an observation that the intuitively strongest attacker (who never forgets anything that has been observed) actually places weaker security demands on the system than we would want. On the other hand they observe that the family of *all* attackers contains pathological attacker behaviours which one certainly does not wish to consider. Due to this they do not give a characterisation of the set of all *reasonable* attackers against which one should protect. We make the following two foundational contributions:

Foundational Contribution 1 We focus (§3.3) on the pragmatic case of *progress insensitive* security (where slow information leakage is allowed through observation of computational progress [AHSS08]). We argue for a new definition of progress insensitive security (Def 11), which unconditionally grants all attackers knowledge of computational progress. With this modification to the definition from [AC12], the problematic examples of pathological attackers are eliminated, and we have a more complete definition of security. Consequently, we are able to prove security in the central contribution of the paper *for all attackers*.

Foundational Contribution 2 The definitions of security are based on characterising attacker knowledge and how it changes over time relative to the changing policy. As argued previously e.g., [BS09], this style of definition forms a much more intuitive basis for a semantics of dynamic policies than using two-run characterisations. However, two-run formulations have the advantage of being easier to use in proofs. We show (§4) that our new knowledge-based progress-insensitive security definition enjoys a simple two-run characterisation. We make good use of this in our proof of correctness of our central contribution.

2 The Dynamic Policy Model

In this section we define an abstract model of computation and a model of dynamic policies which maps computation histories to equivalence relations on stores.

2.1 Computation and Observation Model

Computation Model The computation model is given by a labelled transition system over *configurations*. We write $\langle c, \sigma\rangle \xrightarrow{\alpha} \langle c', \sigma'\rangle$ to mean that configuration $\langle c, \sigma\rangle$ evaluates in one step to configuration $\langle c', \sigma'\rangle$ with label α. Here c is a *command* and $\sigma \in \Sigma$ is a *store*. In examples and when we instantiate this model the store will be a mapping from program variables to values.

The label α records any output that happens during that step, and we have a distinguished label value ϵ to denote a silent step which produces no output. Every non-silent label α has an associated channel $\mathsf{channel}(\alpha) \in Chan$ and a value $\mathsf{value}(\alpha)$. Channels are ranged over by a and values by v. We abbreviate a sequence of evaluation steps

$$\langle c_0, \sigma_0\rangle \xrightarrow{\alpha_1} \langle c_1, \sigma_1\rangle \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \langle c_n, \sigma_n\rangle$$

as $\langle c_0, \sigma_0\rangle \longrightarrow^n \langle c_n, \sigma_n\rangle$. We write $\langle c_0, \sigma_0\rangle \longrightarrow^* \langle c', \sigma'\rangle$ if $\langle c_0, \sigma_0\rangle \longrightarrow^n \langle c', \sigma'\rangle$ for some $n \geq 0$. We write the projection of a single step $\langle c, \sigma\rangle \xrightarrow{\alpha} \langle c', \sigma'\rangle$ to some channel a as $\langle c, \sigma\rangle \xrightarrow{\beta}_a \langle c', \sigma'\rangle$ where $\beta = v$ if $\mathsf{channel}(\alpha) = a$ and $\mathsf{value}(\alpha) = v$, and $\beta = \epsilon$ otherwise, that is, when α is silent or an output on a channel different from a.

We abbreviate a sequence of evaluation steps

$$\langle c_0, \sigma_0\rangle \xrightarrow{\beta_1}_a \langle c_1, \sigma_1\rangle \xrightarrow{\beta_2}_a \ldots \xrightarrow{\beta_n}_a \langle c_n, \sigma_n\rangle$$

as $\langle c_0, \sigma_0\rangle \xrightarrow{t}{}^n_a \langle c_n, \sigma_n\rangle$ where t is the trace of values produced on channel a with every silent ϵ filtered out. We write $\langle c_0, \sigma_0\rangle \xrightarrow{t}_a \langle c', \sigma'\rangle$ if $\langle c_0, \sigma_0\rangle \xrightarrow{t}{}^n_a \langle c', \sigma'\rangle$ for some $n \geq 0$, and we omit the final configuration in contexts where it is not relevant, e.g. $\langle c, \sigma\rangle \xrightarrow{t}_a$. We use $|t|$ to denote the length of trace t.

Attacker's Observation Model We follow the standard assumption that the command c is known to the attacker. We assume a passive attacker which aims to extract information about an input store σ by observing outputs. As in [AC12], the attacker is able only to observe a *single* channel. A generalisation to multi-channel attackers (which would also allow colluding attackers to be modelled) is left for future work.

2.2 Dynamic Policies

A flow policy specifies a limit on how much information an attacker may learn. A very general way to specify such a limit is as an equivalence relation on input stores.

Example 1. Consider a store with variables `x` and `y`. A simple policy might state that the attacker should only be able to learn the value of `x`. It follows that all stores which agree on the value of `x` should look the same to the attacker. This is expressed as the equivalence relation $\sigma \equiv \rho$ iff $\sigma(\mathtt{x}) = \rho(\mathtt{x})$.

A more complicated policy might allow the attacker to learn the value of some arbitrary expression e on the initial store, e.g. $\mathtt{x} = \mathtt{y}$. This is expressed as the equivalence relation $\sigma \equiv \rho$ iff $\sigma(e) = \rho(e)$.

Definition 1 (Policy). *A policy P maps each channel to an equivalence relation $\equiv$ on stores. We write P_a for the equivalence relation that P defines for channel a.*

As defined, policies are static. A dynamic policy changes while the program is running and may dictate a different P for each point in the execution. Here we assume that the policy changes *synchronously* with the execution of the program. That is, the active policy can be deterministically derived from the execution history so far.

Definition 2 (Execution History). *An execution history $\mathcal{H}$ of length n is a transition sequence $\langle c_0, \sigma_0\rangle \xrightarrow{\alpha_1} \langle c_1, \sigma_1\rangle \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \langle c_n, \sigma_n\rangle$.*

Definition 3 (Dynamic Policy). *A dynamic policy D maps every execution history $\mathcal{H}$ to a policy $D(\mathcal{H})$. We write $D_a(\mathcal{H})$ for the equivalence relation that is defined by $D(\mathcal{H})$ for channel a, that is to say, $D_a(\mathcal{H}) = P_a$ where $P = D(\mathcal{H})$.*

Most synchronous dynamic policy languages in the literature determine the current policy based solely on the store σ_n in the final configuration of the execution history [AC12,BvDS13]. Definition 3 allows in principle for more flexible notions of dynamic policies, as they can incorporate the full execution history to determine the policy at each stage of an execution (similar to the notion of conditional noninterference used by [GM84,Zha12]). However, our enforcement does assume that the dynamic policy can be statically approximated per program point, which arguably is only feasible for policies in the style of [AC12,BvDS13]. Such approximations can typically be improved by allowing the program to branch on policy-related queries.

Since programs are deterministic, an execution history of length n is uniquely determined by its initial configuration $\langle c_0, \sigma_0\rangle$. We use this fact to simplify our definitions and proofs:

Definition 4 (Execution Point). *An execution point is a triple (c_0, σ_0, n) identifying the point in execution reached after n evaluation steps starting from configuration $\langle c_0, \sigma_0\rangle$. Such an execution point is considered well-defined iff there exists $\langle c_n, \sigma_n\rangle$ such that $\langle c_0, \sigma_0\rangle \longrightarrow^n \langle c_n, \sigma_n\rangle$.*

Lemma 1. *Each well-defined execution point (c_0, σ_0, n) uniquely determines an execution history $\mathcal{H}(c_0, \sigma_0, n)$ of length n starting in configuration $\langle c_0, \sigma_0\rangle$.*

In the rest of the paper we rely on this fact to justify a convenient abuse of notation, writing $D(c_0, \sigma_0, n)$ to mean $D(\mathcal{H}(c_0, \sigma_0, n))$.

3 Knowledge-Based Security Conditions

Recent works on dynamic policies, including [AC12,BDLG11,BNR08,BS10], make use of so-called *knowledge-based* security definitions, building on the notion of gradual release introduced in [AS07]. This form of definition seems well-suited to provide intuitive semantics for dynamic policies. We focus in particular on the attacker-parametric model from Askarov and Chong in [AC12].

Suppose that the input state to a program is σ. In the knowledge-based approach, an attacker's knowledge of σ is modelled as a *knowledge set* K, which may be any set of states such that $\sigma \in K$. Note that the larger the knowledge set, the less certain is the attacker of the actual value of σ, so a smaller K means more precise knowledge. (Sometimes, as we see below, it can be more intuitive to focus on the complement $\overline{K}$,

which is the set of a-priori possible states which the attacker is able to *exclude*, since this set, which we will call the *exclusion knowledge*, grows as the attacker learns more).

Now suppose that the currently active policy is $\equiv$. The essential idea in any knowledge-based semantics is to view the equivalence classes of $\equiv$ as placing upper bounds on the attacker's knowledge. In the simplest setting, if the actual input state is σ and the attacker's knowledge set is K, we require:

$$K \supseteq \{\sigma' \mid \sigma' \equiv \sigma\}$$

Or, in terms of what the attacker is able to exclude:

$$\overline{K} \subseteq \{\rho \mid \rho \not\equiv \sigma\} \tag{1}$$

How then do we determine the attacker's knowledge? Suppose an attacker knows the program c and observes channel a. Ignoring covert channels (timing, power, etc) an obvious approach is to say that the attacker's knowledge is simply a function of the trace t observed so far:

$$k(t) = \{\rho | \langle c, \rho\rangle \xrightarrow{t}_a\} \tag{2}$$

We define the exclusion knowledge as the complement of this: $ek(t) = \overline{k(t)}$. Note that, as a program executes and more outputs are observed, the attacker's exclusion knowledge can only increase; if $\langle c, \sigma\rangle \xrightarrow{t\cdot v}_a$ then $ek(t) \subseteq ek(t \cdot v)$, since, if ρ can already be excluded by observation of t (because c cannot produce t when started in ρ), then it will still be excluded when $t \cdot v$ is observed (if c cannot produce t it cannot produce any extension of t either).

But this simple model is problematic for dynamic policies. Suppose that the policies in effect when t and $t \cdot v$ are observed are, respectively $\equiv_1$ and $\equiv_2$. Then it seems that we must require both $ek(t) \subseteq \{\rho \mid \rho \not\equiv_1 \sigma\}$ and $ek(t \cdot v) \subseteq \{\rho \mid \rho \not\equiv_2 \sigma\}$. As observed above, it will always be the case that $ek(t) \subseteq ek(t \cdot v)$, so we are forced to require $ek(t) \subseteq \{\rho \mid \rho \not\equiv_2 \sigma\}$. In other words, the observations that we can permit at any given moment are constrained not only by the policy currently in effect but also by all policies which will be in effect in the future. This makes it impossible to have dynamic policies which revoke previously permitted flows (or, at least, pointless; since all revocations would apply retrospectively, the earlier "permissions" could never be exercised).

Askarov and Chong's solution has two key components, outlined in the following two sections.

3.1 Change in Knowledge

Firstly, recognising that policy changes should not apply retrospectively, we can relax (1) to constrain only how an attacker's knowledge should be allowed to *increase*, rather than its absolute value. The increase in attacker knowledge going from t to $t \cdot v$ is given by the set difference $ek(t \cdot v) - ek(t)$. So, instead of (1), we require:

$$ek(t \cdot v) - ek(t) \subseteq \{\rho \mid \rho \not\equiv \sigma\} \tag{3}$$

where $\equiv$ is the policy in effect immediately before the output v. (Some minor set-theoretic rearrangement gives the equivalent

$$k(t \cdot v) \supseteq k(t) \cap \{\sigma' \mid \sigma' \equiv \sigma\}$$

which is the form of the original presentation in [AC12].)

3.2 Forgetful attackers

Focussing on change in knowledge addresses the problem of retrospective revocation but it creates a new issue. Consider the following example.

Example 2. The program in Figure 2 produces the same output many times, but only the first output is permitted by the policy. Assume that the value of x is 5. Before the first output, the exclusion knowledge of an observer on channel a is the empty set. After the first output the observer's exclusion knowledge is increased to include those stores σ where $\sigma(\mathtt{x}) \neq 5$. This is allowed by the policy at that point.

By the time the second output occurs, the policy prohibits any further flows from x. However, since the attacker's exclusion knowledge *already* provides complete knowledge of x, the second output does not actually change the attacker's exclusion knowledge at all, so (3) is satisfied (since $ek(t \cdot v) = ek(t)$). Thus a policy semantics based on (3) would accept this program even though it continues to leak the value of x *long* after the flow has been revoked.

Askarov and Chong address this by revisiting the assumption that an attacker's knowledge is necessarily determined by the simple function of traces (2) above. Consider an attacker which *forgets* the value of the first output in example 2. For this attacker, the second output would come as a revalation, revealing the value of x all over again, in violation of the policy. Askarov and Chong thus arrive at the intriguing observation that security against a more powerful attacker, one who remembers everything that happens, does not imply security against a less resourceful attacker, who might forget parts of the observations made.

```
// x → a
out x on a;
// x ↛ a
while (true)
  out x on a;
```

Fig. 2

Forgetful attackers are modelled as deterministic automata.

Definition 5 (Forgetful Attacker ▷ § III.A [AC12]). *A forgetful attacker is a tuple $A = (S_A, s_0, \delta_A)$ where S_A is the set of attacker states; $s_0 \in S_A$ is the initial state; and $\delta_A : S_A \times Val \to S_A$ the (deterministic) transition function describing how the attacker's state changes due to the values that the attacker observes.*

We write $A(t)$ for the attacker's state after observing trace t:

$$A(\epsilon) = s_0$$
$$A(t \cdot v) = \delta_A(A(t), v)$$

A forgetful attacker's knowledge after trace t is defined as the set of all initial stores that produce a trace which would result in the same attacker state $A(t)$:

Definition 6 (Forgetful Attacker Knowledge ▷ § III.A [AC12]).

$$k(A, c, a, t) = \{\rho \mid \langle c, \rho\rangle \xrightarrow{t'}_a \wedge A(t') = A(t)\}$$

(Note that, in preparation for the formal definition of the security condition, program c and channel a now appear as explicit parameters.)

The proposed security condition is still essentially as given by (3), but now relative to a specific choice of attacker. Stated in the notation and style of the current paper, the formal definition is as follows.

Definition 7 (Knowledge-Based Security ▷ Def. 1 [AC12]). *Command c is secure for policy D against an attacker A on channel a for initial store σ if for all traces t and values v such that $\langle c, \sigma\rangle \xrightarrow{t}{}^{n}_{a} \langle c', \sigma'\rangle \xrightarrow{v}{}^{1}_{a}$ we have*

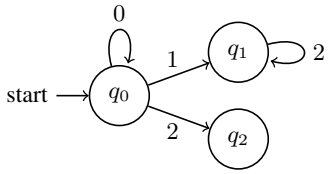
$$ek(A, c, a, t \cdot v) - ek(A, c, a, t) \subseteq \{\rho \mid \rho \not\equiv \sigma\}$$

where $\equiv = D_a(c, \sigma, n)$.

Having relativised security to the power of an attacker's memory, it is natural to consider the strong notion of security that would be obtained by requiring Def. 7 to hold for all choices of A. However, as shown in [AC12], this exposes a problem with the model: there are attackers for which even well-behaved programs are insecure according to Def. 7.

Example 3. Consider again the first example from the Introduction (Section 1.1). Here, for simplicity, we assume that the variable x is boolean, taking value 0 or 1.

```
// x → a
out x on a;
// x ↛ a
out 2 on a;
```



It is intuitively clear that this program complies with the policy. However, as observed in [AC12], if we instantiate Def. 7 with the forgetful attacker displayed, the attacker's exclusion knowledge increases with the second output when $\mathtt{x} = 0$.

After observing the value 0, the attacker's state is $A(0) = q_0$. Since $A(\epsilon) = q_0$, the exclusion knowledge is still the empty set. After the second observation, only stores where $\mathtt{x} = 0$ could have led to state q_2, so the exclusion knowledge increases at a point where the policy does not allow it.

This example poses a question which (so far as we are aware) remains unanswered: if we base a dynamic policy semantics on Def.7, for *which set* of attackers should we require it to hold?

In the next section we define a progress-insensitive variant of Def.7. For this variant it seems that security against all attackers *is* a reasonable requirement and in Section 6 we show that progress-insensitive security against all attackers is indeed enforced by our type system.

3.3 Progress Insensitive Security

Since [VSI96], work on the formalisation and enforcement of information-flow policies has generally distinguished between two flavours of security: *termination-sensitive* and *termination-insensitive*. Termination-sensitive properties guarantee that protected information is neither revealed by its influence on input-output behaviour nor by its influence on termination behaviour. Termination-insensitive properties allow the latter flows and thus provide weaker guarantees. For systems with incremental output (as opposed to batch-processing systems) it is more appropriate to distinguish between *progress-sensitive* and *progress-insensitive* security. Progress-insensitive security ignores progress-flows, where a flow is regarded as a progress-flow if the information that it reveals can be inferred solely by observing *how many* outputs the system produces.

Two examples of programs with progress-flows are as follows:

Example 4. Programs containing progress-flows:

```
// Program A              // Program B
out 1 on a;               out 1 on a;
while (x == 8) skip;      if (x != 8) out 2 on a;
out 2 on a;
```

Let σ and ρ differ only on the value of x: $\sigma(\mathtt{x}) = 4$ and $\rho(\mathtt{x}) = 8$. Note that, if started in σ, both programs produce a trace of length 2 (namely, the trace $1 \cdot 2$) whereas, if started in ρ, the maximum trace length is 1. Thus, for both programs, observing just the length of the trace produced can reveal information about x. Note that, since termination is not an observable event in the semantics, A and B are actually observably equivalent; we give the two variants to emphasise that progress-flows may occur even in the absence of loops.

In practice, most enforcement mechanisms only enforce progress-insensitive security. This is a pragmatic choice since (a) it is hard to enforce progress-sensitive security without being overly restrictive (typically, all programs which loop on protected data will be rejected), and (b) programs which leak solely via progress-flows, leak slowly [AHSS08].

Recall that Knowledge-Based Security (Def. 7) places a bound on the increase in an attacker's knowledge which is allowed to arise from observation of the next output event. Askarov and Chong show how this can be weakened in a natural way to provide a progress-insensitive property, by artificially strengthening the supposed previous knowledge to already include progress knowledge. Their definition of progress knowledge is as follows:

Definition 8 (AC Progress Knowledge ▷ § III.A [AC12]).

$$k^+(A, c, a, t) = \{\rho \mid \langle c, \rho\rangle \xrightarrow{t'\cdot v}_a \wedge A(t') = A(t)\}$$

Substituting this (actually, its complement) in the "previous knowledge" position in Def. 7 provides Askarov and Chong's notion of progress-insensitive security:

Definition 9 (AC Progress-Insensitive (ACPI) Security ▷ Def. 2 [AC12]). *Command c is AC Progress-Insensitive secure for policy D against an attacker A on channel a for initial store σ if for all traces t and values v such that $\langle c, \sigma\rangle \xrightarrow{t}{}^{n}_{a} \langle c', \sigma'\rangle \xrightarrow{v}{}^{1}_{a}$ we have*

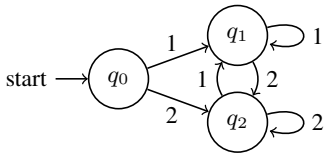
$$ek(A, c, a, t \cdot v) - ek^+(A, c, a, t) \subseteq \{\rho \mid \rho \not\equiv \sigma\}$$

where $\equiv = D_a(c, \sigma, n)$.

Now consider again programs A and B above. These are examples of programs where the *only* flows are progress-flows. In general, we say that a program is *quasi-constant* if there is some fixed (possibly infinite) trace t such that every trace produced by the program is a prefix of t, regardless of the choice of initial store. Thus, for a quasi-constant program, the only possible observable variation in behaviour is trace length, so all flows are progress-flows. Since PI security is intended explicitly to allow progress-flows, we should expect all quasi-constant programs to satisfy PI security, regardless of the choice of policy and for all possible attackers. But, for Def. 9, this fails to hold, as shown by the following counterexample.

Example 5. Consider the program and attacker below. The attacker is a very simple bounded-memory attacker which remembers just the last output seen and nothing else (not even whether it has seen any previous outputs).

```
// x ↛ a
out 1 on a;
out 1 on a;
while (x) skip;
out 1 on a;
out 2 on a;
```



Clearly, the program is quasi-constant. However, it is *not* ACPI secure for the given attacker. To see this, suppose that x = 0 and consider the trace $t = 1 \cdot 1 \cdot 1$. The attacker has no knowledge at this point ($ek(t)$ is the empty set) since it does not know whether it has seen one, two or three 1's. It is easily verified that $ek^+(t)$ is also the empty set for this attacker (intuitively, giving this attacker progress knowledge in the form k^+ doesn't help it, since it still does not know which side of the loop has been reached). But $ek(t \cdot 2)$ is *not* the empty set, since in state q_2 the attacker is able to exclude all stores for which x = 1, thus ACPI security is violated.

What has gone wrong here? The attacker itself seems reasonable. We argue that the real problem lies in the definition of $k^+(A, c, a, t)$. As defined, this is the knowledge that A would have in state $A(t)$ if given just the additional information that c can produce at least one more output. But this takes no account of any *previous* progress knowledge which might have been forgotten by A. (Indeed, the above attacker forgets nearly all such previous progress knowledge.) As a consequence, the resulting definition of PI security mistakenly treats some increases in knowledge as significant, even though they arise purely because the attacker has forgotten previously available progress knowledge.

Our solution will be to re-define progress knowledge to include what the attacker would know *if it had been keeping count*. To this end, for any attacker $A = (S, s_0, \delta)$ we define a counting variant $A^\omega = (S^\omega, s_0^\omega, \delta^\omega)$, such that $S^\omega \subseteq S \times N$, $s_0^\omega = (s_0, 0)$ and $\delta^\omega((s, n), v) = (\delta(s, v), n+1)$. In general, A^ω will be at least as strong an attacker as A:

Lemma 2. *For all A, c, a, t:*

1. $k(A^\omega, c, a, t) \subseteq k(A, c, a, t)$
2. $ek(A, c, a, t) \subseteq ek(A^\omega, c, a, t)$

Proof. It is is easily seen that $A^\omega(t) = (q, n) \Rightarrow A(t) = q$. Thus $A^\omega(t') = A^\omega(t) \Rightarrow A(t') = A(t)$, which establishes part 1. Part 2 is just the contrapositive of part 1.

Our alternative definition of progress knowledge is then:

Definition 10 (Full Progress Knowledge).

$$k^\#(A, c, a, t) = \{\rho \mid \langle c, \rho\rangle \xrightarrow{t' \cdot v}_a \wedge A^\omega(t') = A^\omega(t)\}$$

Our corresponding PI security property is:

Definition 11 (Progress-Insensitive (PI) Security). *Command c is progress-insensitive secure for policy D against an attacker A on channel a for initial store σ if for all traces t and values v such that $\langle c, \sigma\rangle \xrightarrow{t}{}^{n}_{a} \langle c', \sigma'\rangle \xrightarrow{v}{}^{1}_{a}$ we have*

$$ek(A, c, a, t \cdot v) - ek^{\#}(A, c, a, t) \subseteq \{\rho \mid \rho \not\equiv \sigma\}$$

where $\equiv = D_a(c, \sigma, n)$.

This definition behaves as expected for quasi-constant programs:

Lemma 3. *Let c be a quasi-constant program. Then c is PI secure for all policies D against all attackers A on all channels a for all initial stores σ.*

Proof. It suffices to note that, from the definitions, if $t \cdot v$ is a possible trace for c and c is quasi-constant, $ek^{\#}(A, c, a, t) = ek(A^{\omega}, c, a, t \cdot v)$. The result follows by Lemma 2.

As a final remark in this section, we note that there is a class of attackers for which ACPI and PI security coincide. Say that A is *counting* if it always remembers at least how many outputs it has observed. Formally:

Definition 12 (Counting Attacker). *A is counting if $A(t) = A(t') \Rightarrow |t| = |t'|$.*

Now say that attackers A and A' are isomorphic (written $A \cong A'$) if $A(t_1) = A(t_2) \Leftrightarrow A'(t_1) = A'(t_2)$ and note that none of the attacker-parametric security conditions distinguish between isomorphic attackers (in particular, knowledge sets are always equal for isomorphic attackers). It is easily verified that $A \cong A^{\omega}$ for all counting attackers. It is then immediate from the definitions that ACPI security and PI security coincide for counting attackers.

4 Progress-Insensitive Security as a Two-Run Property

Our aim in this section is to derive a security property which guarantees (in fact, is equivalent to) PI security for all attackers, and in a form which facilitates the soundness proof of our type system. For this we seek a property in "two run" form.

First we reduce the problem by establishing that it suffices to consider just the counting attackers.

Lemma 4. *Let c be a command. Then, for any given policy, channel and initial store, c is PI secure against all attackers iff c is PI secure against all counting attackers.*

Proof. Left to right is immediate. Right to left, it suffices to show that

$$ek(A, c, a, t \cdot v) - ek^{\#}(A, c, a, t) \subseteq ek(A^{\omega}, c, a, t \cdot v) - ek^{\#}(A^{\omega}, c, a, t)$$

Since $A^{\omega} \cong (A^{\omega})^{\omega}$, we have $ek^{\#}(A^{\omega}, c, a, t) = ek^{\#}(A, c, a, t)$. It remains to show that $ek(A, c, a, t \cdot v) \subseteq ek(A^{\omega}, c, a, t \cdot v)$, which holds by Lemma 2.

Our approach is now essentially to unwind Def. 11. Our starting point for the unwinding is:

$$ek(A, c, a, t \cdot v) - ek^{\#}(A, c, a, t) \subseteq \{\rho \mid \rho \not\equiv \sigma\}$$

where $\equiv$ is the policy in effect at the moment the output v is produced. Simple set-theoretic rearrangement gives the equivalent:

$$\{\sigma' \mid \sigma' \equiv \sigma\} \cap k^{\#}(A, c, a, t) \subseteq k(A, c, a, t \cdot v)$$

Expanding the definitions, we arrive at:

$$\rho \equiv \sigma \wedge \langle c, \rho\rangle \xrightarrow{t'\cdot v'}_a \wedge\, A^\omega(t') = A^\omega(t) \Rightarrow \exists s. \langle c, \rho\rangle \xrightarrow{s}_a \wedge\, A(s) = A(t \cdot v)$$

By Lemma 4, we can assume without loss of generality that A is counting, so we can replace $A^\omega(t') = A^\omega(t)$ by $A(t') = A(t)$ on the lhs. Since A is counting, we know that $|t| = |t'|$ and $|s| = |t \cdot v|$, hence $|s| = |t' \cdot v'|$. Now, since c is deterministic and both s and $t' \cdot v'$ start from the same ρ, it follows that $s = t' \cdot v'$. Thus we can simplify the unwinding to:

$$\rho \equiv \sigma \wedge \langle c, \rho\rangle \xrightarrow{t'\cdot v'}_a \wedge\, A(t') = A(t) \Rightarrow A(t' \cdot v') = A(t \cdot v)$$

Now, suppose that this holds for A and that $v' \neq v$. Let q be the attacker state $A(t') = A(t)$ and let r be the attacker state $A(t' \cdot v') = A(t \cdot v)$. Since $|t| \neq |t \cdot v|$ and A is counting, we know that $q \neq r$. Then we can construct an attacker A' from A which leaves q intact but splits r into two distinct states r_v and $r_{v'}$. But then security will fail to hold for A', since $A'(t' \cdot v') = r_v \neq r_{v'} = A'(t \cdot v)$. So, since we require security to hold for all A, we may strengthen the rhs to $A(t' \cdot v') = A(t \cdot v) \wedge v = v'$. Then, given $A(t') = A(t)$, since A is a deterministic automaton, it follows that $v = v' \Rightarrow A(t' \cdot v') = A(t \cdot v)$, hence the rhs simplifies to just $v = v'$ and the unwinding reduces to:

$$\rho \equiv \sigma \wedge \langle c, \rho\rangle \xrightarrow{t'\cdot v'}_a \wedge\, A(t') = A(t) \Rightarrow v' = v$$

Finally, since A now only occurs on the lhs, we see that there is a distinguished counting attacker for which the unwinding is harder to satisfy than all others, namely the attacker $A_\#$, for which $A_\#(t') = A_\#(t)$ iff $|t'| = |t|$. Thus the property will hold for all A iff it holds for $A_\#$ and so we arrive at our two-run property:

Definition 13 (Two-Run PI Security). *Command c is two-run PI secure for policy D on channel a for initial store σ if whenever $\langle c, \sigma\rangle \xrightarrow{t}{}^n_a \langle c_n, \sigma_n\rangle \xrightarrow{v}{}^1_a$ and $\rho \equiv \sigma$ and $\langle c, \rho\rangle \xrightarrow{t'\cdot v'}_a$ and $|t'| = |t|$, then $v' = v$, where $\equiv = D_a(c, \sigma, n)$.*

Theorem 1. *Let c be a command. For any given policy, channel and initial store, c is PI secure against all attackers iff c is two-run PI secure.*

Proof. This follows from the unwinding of the PI security definition, as shown above.

5 A Dependency Type System

Within the literature on enforcement of information flow policies, some work is distinguished by the appearance of explicit dependency analyses. In the current paper we take as our starting point the flow-sensitive type systems of [HS11,HS06], due to the relative simplicity of presentation. Other papers proposing similar analyses include [CHH02], [AB04], [AR80] and [BBL94]. Some of the similarities and differences between these approaches are discussed in [HS06].

The original work of [HS06] defines a family of type systems, parameterised by choice of a multi-level security lattice, and establishes the existence of principal typings within this family. The later work of [HS11] defines a single system which produces *only* principal types. In what follows we refer to the particular flow-sensitive type system defined in [HS11] as FST.

Values $v ::= n$ Expressions $e ::= v \mid \mathtt{x}$

Commands $c ::= \mathtt{skip} \mid c_1; c_2 \mid \mathtt{x} := e \mid \mathtt{if}\ e\ c_1\ c_2 \mid \mathtt{while}\ e\ c \mid \mathtt{out}\ e\ \mathtt{on}\ a\ @\ p$

$$\langle \mathtt{skip}; c, \sigma\rangle \xrightarrow{\epsilon} \langle c, \sigma\rangle \qquad \frac{\langle c_1, \sigma\rangle \xrightarrow{\alpha} \langle c_1', \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \xrightarrow{\alpha} \langle c_1'; c_2, \sigma'\rangle} \qquad \frac{\sigma(e) = v}{\langle \mathtt{x} := e, \sigma\rangle \xrightarrow{\epsilon} \langle \mathtt{skip}, \sigma'\rangle}$$

$$\frac{\sigma(e) = v}{\langle \mathtt{out}\ e\ \mathtt{on}\ a\ @\ p, \sigma\rangle \xrightarrow{(a,v,p)} \langle \mathtt{skip}, \sigma'\rangle} \qquad \langle \mathtt{while}\ e\ c, \sigma\rangle \xrightarrow{\epsilon} \langle \mathtt{if}\ e\ (c; \mathtt{while}\ e\ c)\ \mathtt{skip}, \sigma\rangle$$

$$\frac{\sigma(e) \neq 0}{\langle \mathtt{if}\ e\ c_1\ c_2, \sigma\rangle \xrightarrow{\epsilon} \langle c_1, \sigma\rangle} \qquad \frac{\sigma(e) = 0}{\langle \mathtt{if}\ e\ c_1\ c_2, \sigma\rangle \xrightarrow{\epsilon} \langle c_2, \sigma\rangle}$$

Fig. 6: Language and semantics.

The typings derived by FST take the form of an environment Γ mapping each program variable $\mathtt{x}$ to a set $\Gamma(\mathtt{x})$ which has a direct reading as (a conservative approximation to) the set of dependencies for $\mathtt{x}$. All other types derivable using the flow-sensitive type systems of [HS06] can be recovered from the principal type derived by FST. Because principal types are simply dependency sets, they are not specific to any particular security hierarchy or policy. This is the basis of the clean separation we are able to achieve between analysis and policy verification in what follows.

Consider the simple program shown in Figure 5. The type inferred for this program is Γ, where $\Gamma(\mathtt{x}) = \{\}$, $\Gamma(\mathtt{y}) = \{\mathtt{y}, \mathtt{z}\}$, $\Gamma(\mathtt{z}) = \{\mathtt{z}\}$. From this typing we can verify, for example, any static policy using a security lattice in which $level(\mathtt{z}) \sqsubseteq level(\mathtt{y})$.

```
x := z + 1;
z := x;
if (z > 0)
  y := 1;
x := 0;
```

Fig. 5

FST is defined only for a simple language which does not include output statements. This makes it unsuitable for direct application to verification of dynamic policies, so in the current paper we describe a straightforward extenion of FST to a language with output statements. We then show how the inferred types can be used to enforce policies such as those in [AC12] and [BvDS13], which appear very different from the simple static, multi-level policies originally targeted.

5.1 Language

We instantiate the abstract computation model of Section 2.1 with a simple while-language with output channels, shown in Figure 6. We let $\mathtt{x} \in PVar$ range over program variables, $a \in Chan$ range over channels (as before) and $p \in PPoint$ range over program points. Here non-silent output labels have the form (a, v, p), $\mathsf{channel}(a, v, p) = a$, and $\mathsf{value}(a, v, p) = v$.

The language is similar to the one considered in [AC12], except for the absence of input channels. Outputs have to be annotated with a program point p to bridge between the dependency analysis and the policy analysis, described in Section 6.

5.2 Generic typing

Traditional type systems for information flow assume that all sensitive inputs to the system (here: program variables) are associated with a security level. Expressions in the command to be typed might combine information with different security levels. To

ensure that all expressions can be typed, the security levels are therefore required to form at least a join-semilattice, or in some cases a full lattice. The type system then ensures no information of a (combined) level l_1 can be written to a program variable with level l_2 unless $l_1 \sqsubseteq l_2$.

The system FST from Hunt and Sands [HS11] differs from these type systems in two ways. Firstly, it does not require intermediate assignments to respect the security lattice ordering. As an observer is assumed to only see the final state of the program, only the final value of a variable must not depend on any information which is forbidden by the lattice ordering. For example, suppose $level(\mathtt{y}) \sqsubseteq level(\mathtt{z}) \sqsubseteq level(\mathtt{x})$ but $level(\mathtt{x}) \not\sqsubseteq level(\mathtt{z})$ and consider the first two assignments in the example from Fig. 5.

```
x = z + 1; z = x;
```

A traditional type system would label this command as insecure because of the assignment `z = x` and the fact that $level(\mathtt{x}) \not\sqsubseteq level(\mathtt{z})$, even though the value of `z` after this assignment does not depend on the initial value of `x` at all. FST however is *flow-sensitive* and allows the security label on `x` to vary through the code.

Secondly, and more significantly, by using the powerset of program variables as security lattice, FST provides a *principal typing* from which all other possible typings can be inferred.

Thus the typing by FST is generic: a command needs to be typed only once and can then be verified against any static information-flow policy. Since the ordering among labels is not relevant while deriving the typing, FST is also able to verify policies which are not presented in the shape of a security lattice, but any relational *'may-flow'* predicate between security labels can be verified.

5.3 Generic typing for dynamic policies

We now present an extended version of FST which includes an additional typing rule for outputs. All the original typing rules of FST remain unchanged.

Intuitively, an output on a channel is like the final assignment to a variable in the original FST, that is, its value can be observed. Since types are sets of dependencies, we could simply type an output channel as the union of all dependencies resulting from all output statements for that channel. This would be sound but unduly imprecise: the only flows permitted would be those permitted by the policy *at all times*, in effect requiring us to conservatively approximate each dynamic policy by a static one. But we can do better than this.

The flow-sensitivity of FST means that a type derivation infers types at intermediate program points which will, in general, be different from the top-level type inferred for the program. These intermediate types are not relevant for variables, since their intermediate values are not observable. But the outputs on channels at intermediate points *are* observable, and so intermediate channel types *are* relevant. Therefore, for each channel we record in Γ distinct dependency sets for each program point at which an output statement on that channel occurs. Of course, this is still a static approximation of runtime behaviour. While our simple examples of dynamic policies explicitly associate policy changes to program points, for real-world use more expressive dynamic policy languages may be needed. In Section 2.2 we formally define the semantics of a dynamic policy as an arbitrary function of a program's execution history, which provides a high

degree of generality. However, in order to apply a typing to the verification of such a policy, it is first necessary to conservatively approximate the flows permitted by the policy at each program point of interest (Definition 16).

Let X be the dependency set for the channel-a output statement at program point p. The meaning[3] of X is as follows:

> Let σ be a store such that execution starting in σ arrives at p, producing the i'th output on a. Let ρ be any store which agrees with σ on all variables in X and also eventually produces an i'th output on a (not necessarily at the same program point). Then these two outputs will be equal.

Two key aspects of our use of program points should be highlighted:

1. While the intended semantics of X as outlined above does not require corresponding outputs on different runs to be produced at the same program point, the X that is inferred by the type system *does* guarantee this stronger property. Essentially this is because (in common with all similar analyses) the type system uses control-flow dependency as a conservative proxy for the semantic dependency property of interest.
2. Our choice of program point to distinguish between different ouputs on the same channel is not arbitrary; it is essentially forced by the structure of the original type system. As noted, program point annotations simply allow us to record in the final typing exactly those intermediate dependency sets which are already inferred by the underlying flow-sensitive system. While it would be possible in principle to make even finer distinctions (for example, aiming for path-sensitivity rather than just flow-sensitivity) this would require fundamental changes to the type system.

The resulting type system is shown in Figure 7. We now proceed informally to motivate its rules. Definitions and proofs of formal soundness are presented in Section 6.

The type system derives judgements of the form $\vdash\{c\}\Gamma$, where $\Gamma : \mathit{Var} \to 2^{\mathit{Var}}$ is an environment mapping variables to a set of dependencies. The variables we consider are $\mathit{Var} = \mathit{PVar} \cup \mathit{CPoint} \cup \{\mathtt{pc}\} \cup \mathit{Chan}$ with $\mathit{CPoint} = \mathit{Chan} \times \mathit{PPoint}$. We consider the relevance of each kind of variable in turn.

- As program variables PVar form the inputs to the command, these are the dependencies of interest in the typing of a command. For program variables themselves, $\Gamma(\mathtt{x})$ are the dependencies for which a different initial value might result in a different final value of $\mathtt{x}$.
- Pairs of channels and program points $(a, p) \in \mathit{CPoint}$ are denoted as a_p. The dependencies $\Gamma(a_p)$ are those program variables for which a difference in initial value might cause a difference in the value of any observation that can result from an output statement for channel a with annotation p.
- Whenever the program counter $\mathtt{pc} \in \Gamma(\mathtt{x})$ this indicates that this command potentially changes the value of program variable $\mathtt{x}$. Similar, if $\mathtt{pc} \in \Gamma(a)$ then c might produce an output on channel a and if $\mathtt{pc} \in \Gamma(a_p)$ then c might produce an output

[3] This is progress-insensitive dependency (see Section 3). A progress-sensitive version can be defined in a similar way.

$$\text{TS-SKIP}\quad \vdash \{\texttt{skip}\}\ \Gamma_{id} \qquad\qquad \text{TS-ASSIGN}\quad \vdash \{\texttt{x} := e\}\ \Gamma_{id}\ [\texttt{x} \mapsto fv(e) \cup \{\texttt{pc}\}]$$

$$\text{TS-SEQ}\quad \frac{\vdash \{c_1\}\Gamma_1 \qquad \vdash \{c_2\}\Gamma_2}{\vdash \{c_1\ ;\ c_2\}\ \Gamma_2;\Gamma_1}$$

TS-IFELSE

$$\frac{\vdash \{c_i\}\Gamma_i \qquad \vdash \Gamma_i' = \Gamma_i;\Gamma_{id}[\texttt{pc} \mapsto \{\texttt{pc}\} \cup fv(e)] \qquad i = 1,2}{\vdash \{\texttt{if}\ e\ c_1\ c_2\}\ (\Gamma_1' \cup \Gamma_2')[\texttt{pc} \mapsto \{\texttt{pc}\}]}$$

TS-WHILE

$$\frac{\vdash \{c\}\Gamma_c \qquad \Gamma_f = (\Gamma_c;\Gamma_{id}[\texttt{pc} \mapsto \{\texttt{pc}\} \cup fv(e)])^*}{\vdash \{\texttt{while}\ e\ c\}\ \Gamma_f\ [\texttt{pc} \mapsto \{\texttt{pc}\}]}$$

TS-OUTPUT

$$\vdash \{\texttt{out}\ e\ \texttt{on}\ a\ @\ p\}\Gamma_{id}[a_p \mapsto fv(e) \cup \{\texttt{pc}, a, a_p\}; a \mapsto \{\texttt{pc}, a\}]$$

Fig. 7: Type System.

on a caused by a statement annotated with p. We use the program counter to catch implicit flows that may manifest in these ways.

– We use $Chan$ to capture the latent flows described in example program B in the introduction. The dependencies $\Gamma(a)$ are those program variables for which a difference in initial value might result in a different number of outputs produced on channel a by this command. This approach to address latent flows was first introduced in [AC12] as *channel context bounds*.

We first explain the notation used in the unchanged rules from FST before turning our attention to the new TS-OUTPUT rule. All concepts have been previously introduced in [HS11].

The function $fv(e)$ returns the free variables in expression e. The identity environment Γ_{id} maps each variable to the singleton set of itself, that is $\Gamma_{id}(x) = \{x\}$ for all $x \in Var$. Sequential composition of environments is defined as:

$$\Gamma_2;\Gamma_1(x) = \bigcup_{y \in \Gamma_2(x)} \Gamma_1(y)$$

Intuitively, $\Gamma_2;\Gamma_1$ is as Γ_2 but substituting the dependency relations already established in Γ_1. We overload the union operator for environments: $(\Gamma_1 \cup \Gamma_2)(x) = \Gamma_1(x) \cup \Gamma_2(x)$. We write Γ^* for the fixed-point of Γ, used in TS-WHILE:

$$\Gamma^* = \bigcup_{n \geq 0} \Gamma^n \qquad \text{where } \Gamma^0 = \Gamma_{id} \text{ and } \Gamma^{n+1} = \Gamma^n;\Gamma$$

It is only in the typing TS-OUTPUT of the output command that the additional channel and program point dependencies are mentioned; this underlines our statement that extending FST to target dynamic policies is straightforward.

We explain the changes to Γ_{id} in TS-OUTPUT in turn. For a_p, clearly the value of the output and thus the observation is affected by the program variables occuring in the expression e. We also include the program counter pc to catch implicit flows; if we have

a command of the form $\texttt{if}\ e\ (\texttt{out}\ 1\ \texttt{on}\ a\ @\ p)\ (\texttt{out}\ 2\ \texttt{on}\ a\ @\ q)$ the output at a_p is affected by the branching decision, which is caught in TS-IFELSE.

We include the channel context bounds a for the channel on which this output occurs to capture the latent flows of earlier conditional outputs, as demonstrated in the introduction. Observe that by the definition of sequential composition of environments, we only add those dependencies for conditional outputs that happened *before* this output, since it cannot leak information about the absence of future observations.

Finally, we include the dependencies of output point a_p itself. By doing so the dependency set of a_p becomes *cumulative*: with every sequential composition (including those used in Γ^*) the dependency set of a_p only grows, as opposed to the dependencies of program variables. This makes us sum the dependencies of all outputs on channel a annotated with the same program point, as we argued earlier.

The mapping for channel context bounds a is motivated in a similar manner. The pc is included since the variables affecting whether this output occurs on channel a are the same as those that affect whether this statement is reached. Note that we are over-approximating here, as the type system adds the dependencies of e in

$$\texttt{if}\ e\ (\texttt{out}\ 1\ \texttt{on}\ a\ @\ p_1)\ (\texttt{out}\ 2\ \texttt{on}\ a\ @\ p_2)$$

to context bounds a, even though the number of outputs is always one.

Like for a_p, we make a depend on itself, thus accumulating all the dependencies that affect the number of outputs on channel a.

As the TS-OUTPUT rule does not introduce more complex operations than already present, the type system has the same complexity as FST. That is, the type system can be used to construct a generic type in $O(nv^3)$ where n is the size of the program and v the number of variables in *Var*.

6 Semantic Soundness and Policy Compliance

We present a soundness condition for the type system, and show that the type system is sound. We then describe how the derived generic typings can be used to check compliance with a dynamic policy that is approximated per program point. We begin by showing how an equivalence relation on stores can be created from a typing:

Definition 14. *We write $=_{\Gamma(x)}$ for the equivalence relation corresponding to the typing Γ of variable $x \in Var$, defined as $\sigma =_{\Gamma(x)} \rho$ iff $\sigma(\mathtt{y}) = \rho(\mathtt{y})$ for all $\mathtt{y} \in \Gamma(x)$.*

As we are using $\Gamma(a_p)$ as the approximation of dependencies for an observation, the soundness of the PI type system is similar to the PI security for dynamic policies, except that we take the equivalence relation as defined by $\Gamma(a_p)$ rather than the policy D_a.

Definition 15 (PI Type System Soundness). *We say that the typing $\vdash\{c\}\Gamma$ is sound iff for all σ, ρ, if $\langle c, \sigma\rangle \xrightarrow{t}_a \langle c_\sigma, \sigma'\rangle \xrightarrow{(a,v,p)}$ and $\langle c, \rho\rangle \xrightarrow{t'}_a \langle c_\rho, \rho'\rangle \xrightarrow{v'}_a$ and $|t| = |t'|$ then $\sigma =_{\Gamma(a_p)} \rho \Rightarrow v = v'$.*

Theorem 2. *All typings derived by the type system are sound.*

The proof for Theorem 2 can be found in Appendix A of [DHS15].

To link the typing and the actual dynamic policy, we rely on an analysis that is able to approximate the dynamic policy per program point. A sound approximation should

return a policy that is at least as restrictive as the actual policy for any observation on that program point.

Definition 16 (Dynamic Policy Approximation). *A dynamic policy approximation $A : CPoint \rightarrow 2^{\Sigma \times \Sigma}$ is a mapping from channel and program point pairs to an equivalence relation on stores. The approximation A on command c, written $c : A$, is sound for dynamic policy D iff, for all σ if $\langle c, \sigma \rangle \rightarrow^n \langle c', \sigma' \rangle \xrightarrow{(a,v,p)}$ then $A(a_p)$ is coarser than $D_a(c, \sigma, n)$.*

We now arrive at the main theorem in this section. Given a typing $\vdash \{c\}\Gamma$, we can now easily verify for command c its compliance with *any* soundly approximated dynamic policy, by simply checking that the typing's policy is at least as restrictive as the approximated dynamic policy for every program point.

Theorem 3 (PI Dynamic Policy Compliance). *Let $c : A$ be a sound approximation of dynamic policy D. If $\vdash \{c\}\Gamma$ and $=_{\Gamma(a_p)}$ is coarser than $A(a_p)$ for all program points a_p, then c is two-run PI secure for D on all channels and for all initial stores.*

Proof. Given a store σ such that $\langle c, \sigma \rangle \xrightarrow{t}{}^n_a \langle c_\sigma, \sigma' \rangle \xrightarrow{(a,v,p)}$ and a store ρ such that $\langle c, \rho \rangle \xrightarrow{t'}{}_a \langle c_\rho, \rho' \rangle \xrightarrow{v'}{}_a$ and $|t| = |t'|$ and $\sigma D_a(c, \sigma, n)\rho$, we need to show that $v = v'$. Since $c : A$ is a sound approximation of D, we have that $\sigma A(a_p)\rho$ and as $=_{\Gamma(a_p)}$ is coarser than $A(a_p)$ we also have $\sigma =_{\Gamma(a_p)} \rho$. Which by Theorem 2 gives us that $v = v'$.

Corollary 1. *If the conditions of Theorem 3 are met, then c is PI secure for D for all attackers. This is immediate by Theorem 1.*

7 Related Work

In this section we consider the related work on security for dynamic policies and on generic enforcement mechanisms for information-flow control. We already discuss the knowledge-based definitions by Askarov and Chong [AC12] in detail in Section 3.

The generality of expressing dynamic policies per execution point can be identified already in the early work by Goguen and Meseguer [GM82]. They introduce the notion of conditional noninterference as a relation that should hold per step in the system, provided that some condition on the execution history holds. Conditional noninterference has been recently revisited by Zhang [Zha12] who uses unwinding relations to present a collection of properties that can be verified by existing proof assistants.

Broberg and Sands [BS09] developed another knowledge-based definition of security for dynamic policies which only dealt with the attacker with perfect recall. The approach was specialised to the specific dynamic policy mechanism Paralocks [BS10] which uses part of the program state to vary the ordering between security labels.

Balliu et al. [BDLG11] introduce a temporal epistemic logic to express information flow policies. Like our dynamic policies, the epistemic formulas are to be satisfied per execution point. Dynamic policies can be individually checked by the ENCoVER tool [BDLG12].

The way in which we define dynamic policies matches exactly the set of synchronous dynamic policies: those policies that deterministically determine the active policy based on an execution point. Conversely, an asynchronously changing policy

cannot be deterministically determined from an execution point, but is influenced by an environment external to the running program.

There is relatively little work on the enforcement of asynchronous dynamic policies. Swamy et al. [SHTZ06] present the language RX where policies are define in a role-based fashion, where membership and delegation of roles can change dynamically. Hicks et al. [HTHZ05] present an extension to the DLM model, allowing the acts-for hierarchy among principals to change while the program is running.

Both approaches however need a mechanism to synchronise the policy changes with the program in order to enforce information-flow properties. RX uses transactions which can rollback when a change in policy violates some of the flows in it, whereas the work by Hicks et al. inserts automatically derived coercions that force run-time checks whenever the policy changes.

A benefit of our enforcement approach is that commands need to be analysed only once to be verified against multiple information-flow policies. This generality can also be found in the work by Stefan et al. [SRMM11] presenting LIO, a Haskell library for information-flow enforcement which is also parametric in the security policy. The main differences between our approach and theirs is that LIO's enforcement is dynamic rather than static, while the enforced policies are static rather than dynamic.

8 Conclusions

We extended the flow-sensitive type system from [HS06] to provide for each output channel individual dependency sets per point in the program and demonstrated that this is sufficient to support dynamic information flow policies. We proved the type system sound with respect to a straightforward two-run property which we showed sufficient to imply knowledge-based security conditions.

As our approach allows for the core of the analysis to be performed even before the policy is known, this enables us to reuse the results of the dependency analysis across the verification of multiple types of policies. An interesting direction for future research could be on the possibility to use the dependency analyses performed by advanced slicing tools such as JOANA [JOA] and Indus [Ind].

Acknowledgements This work is partly funded by the Swedish agencies SSF and VR.

References

[AB04] T. Amtoft and A. Banerjee. Information Flow Analysis in Logical Form. In *11th Static Analysis Symposium*, volume 3148 of *LNCS*, pages 100–115. Springer, 2004.

[AC12] A. Askarov and C. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Computer Security Foundations Symposium*, pages 308–322. IEEE, 2012.

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *The 13th European Symposium on Research in Computer Security*, number 5283 in LNCS, pages 333–348. Springer, 2008.

[AR80] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[AS07] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[BBL94] J.-P. Banâtre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *Proc. European Symp. on Research in Computer Security*, volume 875 of *LNCS*, pages 55–73. Springer-Verlag, 1994.

[BDLG11] M. Balliu, M. Dam, and G. Le Guernic. Epistemic temporal logic for information flow security. In *Programming Languages and Analysis for Security*, PLAS '11, pages 6:1–6:12. ACM, 2011.

[BDLG12] M. Balliu, M. Dam, and G. Le Guernic. Encover: Symbolic exploration for information flow security. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 30–44. IEEE, 2012.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353. IEEE Computer Society, 2008.

[BS09] N. Broberg and David S. Flow-Sensitive Semantics for Dynamic Information Flow Policies. In *Programming Languages and Analysis for Security*, 2009.

[BS10] N. Broberg and D. Sands. Paralocks – role-based information flow control and beyond. In *Symposium on Principles of Programming Languages*. ACM, 2010.

[BvDS13] N. Broberg, B. van Delft, and D. Sands. Paragon for Practical Programming with Information-Flow Control. In *Programming Languages and Systems*, volume 8301 of *LNCS*, pages 217–232. 2013.

[CHH02] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *Journal of Computer Languages*, 28(1):3–28, April 2002.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[DHS15] B. van Delft, S. Hunt, and D. Sands. Very Static Enforcement of Dynamic Policies. Technical Report 1501.02633, arXiv, 2015.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, April 1984.

[HS06] S. Hunt and D. Sands. On Flow-sensitive Security Types. In *Symposium on Principles of Programming Languages*, pages 79–90. ACM, 2006.

[HS11] S. Hunt and D. Sands. From Exponential to Polynomial-time Security Typing via Principal Types. In *Programming Languages and Systems. ESOP*, number 6602 in LNCS. Springer Verlag, 2011.

[HTHZ05] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Foundations of Computer Security Workshop*, pages 7–18, 2005.

[Ind] Indus homepage. http://indus.projects.cis.ksu.edu/. Accessed: 2015-01-09.

[JOA] JOANA homepage. http://pp.ipd.kit.edu/projects/joana/. Accessed: 2015-01-09.

[SHTZ06] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing Policy Updates in Security-Typed Languages. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, 2006.

[SRMM11] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, 2011.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[Zha12] C. Zhang. Conditional Information Flow Policies and Unwinding Relations. In *Trustworthy Global Computing*, volume 7173 of *LNCS*, pages 227–241. Springer, 2012.