2015 IEEE 28th Computer Security Foundations Symposium

The Anatomy and Facets of Dynamic Policies

Niklas Broberg Bart van Delft David Sands
Chalmers University of Technology, Sweden

***Abstract*—Information flow policies are often dynamic; the security concerns of a program will typically change during execution to reflect security-relevant events. A key challenge is how to best specify, and give proper meaning to, such dynamic policies. A large number of approaches exist that tackle that challenge, each yielding some important, but unconnected, insight. In this work we synthesise existing knowledge on dynamic policies, with an aim to establish a common terminology, best practices, and frameworks for reasoning about them. We introduce the concept of *facets* to illuminate subtleties in the semantics of policies, and closely examine the *anatomy* of policies and the *expressiveness* of policy specification mechanisms. We further explore the relation between dynamic policies and the concept of declassification.**

I. INTRODUCTION

Many of the security concerns that arise in software can be understood in terms of information flows. For example, confidentiality requires that sensitive data does not flow to places where it can be observed by unauthorised subjects, whereas integrity requires that untrusted data does not flow to places which are trusted by other components in the system. Although information flow policies can express these concerns, in practice the desired information flows that a system permits is not a static notion, but one which may change during the running of the system to reflect security-relevant events. For example, information under a restrictive information flow policy may be less restricted if the information has been purchased. Conversely, the privileges of a principal may be revoked, thus reducing the information flows deemed acceptable. We refer to information flow policies that may change during a single instance of a system as *dynamic information flow policies*, or simply *dynamic policies*.

Over the last twenty years a large number of systems and mechanisms presented answers to questions as “How can we express policies that let us specify which flows are acceptable?” and “How can we express semantic properties that precisely and succinctly capture what it means for a program not to contain unintended leaks?”. With this plethora of answers, another set of questions naturally arise: which are the *best* answers, in a given context or for a specific task? By which means can we compare such systems and mechanisms to one another? Which policy specification mechanisms are suitably expressive to capture the security requirements of a specific use case? What proposed semantic property gives strong enough guarantees against a particular kind of attacker?

To these new *meta*-questions, we have far fewer answers. With a few exceptions, what we have for guidance are *case studies* that compare systems in the light of one specific context or task [3,24,32,38]. While certainly useful and valuable for the insights they do give, case studies are by nature not suited for drawing general conclusions or establishing principles.

This paper also provides no distinct answers to these questions; rather it aims to give guidance to those who seek to answer them by providing a synthesis of knowledge on dynamic policies. Our over-arching aim and contribution is to provide a clearer picture of information flow control in the presence of dynamic policies, to facilitate understanding, defining, analysing, comparing, and discussing properties and mechanisms. One of our key contributions, running as a theme through the paper, is therefore to establish and argue for a *common vocabulary*, and to identify *common best-practices* in how we work with dynamic information-flow policies.

Our study explores three main areas: how policies are given semantics; a classification of the types of dynamic policies that exist; and finally a reflection on the relationship to the concept of declassification. We summarise the contributions of each of these in turn.

Semantic Specification of Dynamic Policies (§III) What is the semantic meaning of a dynamic policy, and what are the best practices for specifying them? To give a semantics to an information flow policy one must define a security property: what does it means for a program to *satisfy the given policy*? In other words, when are the information flows that occur when executing a program permitted by the policy? When a program satisfies a given policy we say that the program is *secure* (with respect to that policy). But giving a semantics to a dynamic policy can be subtle, complicated, and often unintuitive. It turns out, as we will show, that small and seemingly minor changes to the details of a security property can lead to fundamental differences in which programs are considered semantically secure, differences that were previously poorly studied and understood. To help put focus on these differences, we introduce the general concept of *facets* of dynamic security properties. A facet captures a type of information flow which is permitted by some definitions but not by others. We identify a number of such facets, argue for how they can be approached, and review how the literature treats these facets.

Classification of Dynamic Policies (§IV) Dynamic policies can be specified in a variety of ways, offering a variety of expressiveness. In the second part of our synthesis we take a closer look at policy specification mechanisms. We explain the general *anatomy* of a dynamic policy in terms of a three-level *hierarchy of control*. Level 0 control refers to the ability of a policy mechanism to specify *which flow relations* can arise in a dynamic policy. Level 1 control refers to the way a mechanism controls *how* the policy changes during execution. Level 2 control refers to the meta-policy, i.e. how a mechanism controls *which policy changes* may occur. From this hierarchy we derive a framework for formal comparisons of the expressiveness of policy specification mechanisms, in terms of what *invariants* they can enforce. For example, can a given mechanism represent an invariant such as “flows from top-secret to secret are never allowed”? Can a mechanism enforce that the permitted information flows always increase (or always decrease) over

DOI 10.1109/CSF.2015.16

time? On top of our hierarchy we apply the "dimensions of declassification" introduced by Sabelfeld and Sands [36] to shed new light on the actual meaning and influence of these "dimensions" on dynamic policies.

Declassification (§V) Finally, we reflect on the concept of "declassification" and its relation to dynamic policies. We look at two different flavours of declassification – *relabelling* and *copying release* – and discuss how these can be interpreted and explained through dynamic policies. Further we identify relationships between support for declassification, and the choices we make for certain facets of security properties.

II. TERMINOLOGY

Much of the terminology used in this paper is overloaded and used inconsistently across the literature. Here we fix the basic terminology used in this paper, with the additional explicit purpose to establish a common vocabulary.

An *information flow policy* refers to a specification of the information flows which are permitted during program execution. At any given point, the permitted flows are given by a *flow relation*. This, for example, might be a specification that input variable x is allowed to flow to output channel y. When the permitted information flows do not change over time, i.e. a single flow relation is used exclusively throughout computation, we refer to the policy as a *static policy*. When the permitted flows (may) change during computation, we call it a *dynamic policy*. A dynamic policy is thus a specification of a set of flow relations, any one of which is active at a given point in time, together with a specification of how the system transitions between them.[1]

Information flow policies are often specified indirectly via a set of *labels*. For example, the classic static two-level security setting is defined by a set of two labels $\{L, H\}$ (for low and high confidentiality, respectively). A *flow relation* is then described by (i) assigning labels to appropriate sources and sinks in a configuration, and (ii) defining a relation between labels, indicating allowed flows. For example in the classic two-level setting, secret inputs are labelled H, and public outputs are labelled L, and the flow policy is specified by saying that the may-flow relation is the smallest reflexive relation such that L may flow to H, which we write as $L \to H$.

A *policy scheme* is a set of labels, their flow relations and transitions, viewed in isolation from the label assignment in any particular program or system. In a policy scheme, a flow relation therefore only consists of the relation between labels (point (ii) above). Finally, different approaches provide different mechanisms (languages) with which to construct such policy schemes, and different ways to assign labels to relevant entities. We refer to these as a *policy specification mechanism*.

III. SEMANTICS OF SECURITY PROPERTIES

Non-interference is the name usually given to the semantic definition of when a program (or system) satisfies a *static* policy. In the absence of subtleties arising from non-determinism and interaction, non-interference could be argued to be a simple and intuitive property, easily stated in the following or any one of several equivalent forms: If information from input i is not allowed to flow to output o, then no variety in i may result in a different output o [20].

Even in the settings where there exist various forms of non-interference, they are often related according to various trade-offs between security and permissiveness [21,43]. However, once we abandon static policies the situation changes dramatically. Not only does each new mechanism for specifying dynamic policies introduce a new semantic definition, the definitions are typically formulated in completely new styles with different attacker models. How these models relate to one another is not easy to see even with extensive scrutiny. Nor is it intuitively clear at first glance just what notion of security a particular property actually guarantees.

In the case of dynamic policies we believe that different systems have more fundamentally different semantic requirements, and different notions of security and attacks. Our grand quest then changes, from one of finding *the* security property, to that of identifying how best to state a security property for a specific set of requirements.

In this section we discuss various aspects of the art of defining information flow security properties. We do not propose any new security properties, instead we identify and illuminate relevant principles and *facets* that affect such properties, and suggest practical consequences and interpretations of the various choices that can be made. Furthermore, we give a survey of previously proposed properties from other work and show how they fall within the spectrum of our facets.

Our purpose here is two-fold: firstly, we hope to provide tools for comparing and reasoning about security properties, to more easily understand how they relate or differ. Secondly, we want to provide the would-be property author with a toolbox of best practices and relevant concerns, to assist in stating a property suitable for that author's specific context.

This section will proceed as follows: First, we argue strongly for why so-called knowledge-based, or *epistemic*, formulations of properties are preferable to the traditional two-run-style formulations. Attempts at such arguments have been made previously [4,5,12], but in passing and never with a complete and coherent picture. Second, we identify a number of *facets* of information flow security properties. We give illustrating examples, and discuss how previously proposed semantic properties differ, sometimes subtly, in the choices made. As we will show, typically no universally "best" choice exists for these facets, which further emphasises the futility of trying to find a single property that works for all situations. Finally we categorise existing work according to the facets we identify.

A. Epistemic information flow security

Askarov and Sabelfeld introduced the *Gradual release* property [4] for information flow control using an explicit model of *attacker knowledge evolution* (also called an *epistemic* property) when observing a single program run. Askarov and Sabelfeld drew inspiration from work on *deducibility* [4], which in turn borrowed the technique from work on

---

[1] Observe the difference between a dynamic policy and a *dynamic label*. The latter refers to a security label which is only known at run time. The label itself however does not change during execution, neither does its relation to other labels (see e.g. [46]).

possibilistic security [23,43]. Some later systems have used epistemic formulations [5,7,9,12].

The main purpose of this section is to argue that epistemic formulations are attractive in that they capture the desired property in a natural and direct fashion. We wish to be clear that the definitions in this section are not essentially novel, but generalised versions of those presented in the cited literature.

The core feature of an epistemic formulation of security is that we consider the system from the perspective of an attacker's knowledge of that system, and how observations on the system change this knowledge. For some observation, we refer to the information that is allowed to be revealed at the moment that observation is produced as the *release policy* (terminology cf. Balliu [7]), denoted $\mathcal{R}_{now}$, and to the attacker's knowledge just before and after this observation as $\mathcal{K}_{before}$ and $\mathcal{K}_{now}$. With these simple terms we can present the general form of an epistemic security property:

$$\mathcal{R}_{now} \text{ allows } increase(\mathcal{K}_{before}, \mathcal{K}_{now})$$

We now continue by specifying a concrete computational system and what is meant with knowledge, although different models could have been used. We say that the execution of a program yields a *trace* of program *events*. We deliberately remain abstract in the nature of the events themselves. For a program P and an initial memory store S, we denote with $\langle P, S\rangle \xrightarrow{t}$ that t is a prefix of the event trace produced by P. Let $obs_A(t)$ be a function determining what observation the attacker A makes on the event trace t. Again we remain abstract to what these observations are, but options include the series of values output on A's channel and updates to memory locations observable by A. It also possible to have more complicated observation functions, an example of which can be found in the work by Askarov and Chong [5].

We model the inputs to the system as the initial memory. The attacker's knowledge can thus be expressed as how much the attacker knows about these initial values in the memory, based on the observations made. In existing work such as Gradual Release, this is presented as the set of initial memory stores for which the program can produce the observation made by A. That is, if $\langle P, S\rangle \xrightarrow{t}$ with $obs_A(t) = o$, the attacker A's knowledge on S is:

$$k_A(P, o) = \{S' | \langle P, S'\rangle \xrightarrow{t'} \text{ and } obs_A(t') = o\}$$

To arrive at a more natural definition of knowledge *increase* we follow the example of van Delft et al. [42] and work instead with the complementary notion of *exclusion knowledge* – the set of initial memories that could *not* have led to the observed trace, i.e. the set complement of k_A:

$$ek_A(P, o) = \overline{k_A(P, o)}$$

Knowledge gain can then conveniently be expressed as the set of stores that the attacker *additionally* excludes after making a next observation. Let $\langle P, S\rangle \xrightarrow{t \cdot e}$ denote that event e was preceded by the trace of events t. A's increase in knowledge is described as the difference between A's knowledge before and after event e:

$$ek_A(P, obs_A(t \cdot e)) \setminus ek_A(P, obs_A(t))$$

where $\setminus$ is the set difference operator. Finally, the release policy that specifies the *permitted* increase in knowledge can be expressed as an *upper bound* on the knowledge gained:

$$ek_A(P, obs_A(t \cdot e)) \setminus ek_A(P, obs_A(t)) \subseteq \mathcal{R}_{now}$$

where $\mathcal{R}_{now}$ is also a set of memory stores. For example, in the setting of two-level non-interference we want to forbid the attacker from learning anything about the secrets in the initial store. Hence, non-interference can be encoded by making $\mathcal{R}_{now}$ the set of stores that have different values from the actual initial store S on the locations not containing secrets.

Depending on the policy language used, $\mathcal{R}_{now}$ can be parameterised over a wide range of aspects – in particular the attacker A and the flow relation dictated by the dynamic policy when e was produced. Other possible aspects include the event e itself, the trace t, the current memory store at the time of the observation, the current program at the time of the observation, or some entirely external entity, for example the system clock.

In words, this abstract property can be expressed as "an attacker observing an event produced by a running program cannot learn anything about the initial memory that is not allowed by the policy at the point of the event".

We posit that this very simple abstract formulation captures the vast majority of properties we want to express through different instantiations of observation models $obs_A(\cdot)$, release policies $\mathcal{R}_{now}$ and observable traces of events $t \cdot e$. But while this abstract form is quite simple, instantiating it properly can be quite tricky.

Why epistemic? Consider a two-run formulation of non-interference for a deterministic program. If spelled out in words, it would read something like (disregarding termination): "if the program is run twice with the same public inputs but possibly different secrets, the public outputs must be the same in both runs". We treat the concepts of inputs, outputs and the notion of public informally here, what is important is that the above quote conveys the gist of such a two-run formulation. Now consider how one would convince a non-expert that this is indeed a suitable characterisation of a program that does not leak any secret information. That argument would very likely be something like: "if this holds, then an attacker observing the outputs of running the program could not deduce anything about what the secret inputs are." But that is exactly what the epistemic property states! The notion of security intrinsically has nothing to do with observing two separate runs – but rather what can be deduced from observing a single run.

The above argument strongly suggests that the epistemic formulation is the most natural way we can state the desired property. A two-run formulation could certainly be very useful as part of the strategy to *prove* e.g. the correctness of an enforcement mechanism; often a (mechanical) two-run formulation can lend itself well to the structure of a proof over execution traces. But that property is then only a stepping stone, and should, for completeness, be shown to imply the natural epistemic property.

Another, purely technical reason to prefer epistemic formulations over *bisimulation-based* two-run properties is pointed out by [12]: the latter are often overly conservative. We return to this point in section III-C.

B. Facets of Semantic Security Properties

When defining an information flow security property, choices are made that ultimately affect what programs are considered secure according to that property. These choices determine what we call the *facets of security properties*.

Facet

An aspect of a security property that determines whether a particular class of information flows is accepted as secure.

Note: The aim of this section is not to formally define these particular classes of information flows. Rather, we point out several examples that underline the existence of facets as a design space for security properties.

Various effects could be regarded as facets of a security property. A well-known facet is *termination sensitivity*: a property can either allow programs to leak sensitive information through their termination behaviour, or not. This is a facet of which current designers of security properties (and enforcement mechanisms) are well aware. A conscious choice is made if this facet is addressed, often motivated from the pragmatic perspective that an enforcement for a termination insensitive property is easier to achieve and less restrictive. More facets of security properties exist, but their existence is largely unknown and hence how a property treats these facets is not by conscious choice.

In this paper we are interested in those facets that arise in various semantics properties of dynamic policies. Each facet is not universal among the works that we have surveyed – some definitions permit a facet while at least one other does not. But in most cases the *choice* for each facet is far from explicit. We attempt to present a justification for both the permissive as well as the restrictive treatment of each facet, although arguably these pleas are not equally compelling in all cases.

Our main contribution of this section is *not* to identify these facets – some facets were identified before, and we make no claim that we have identified all facets – but to argue for recognising the *existence* of facets and provide the designer of a security condition with the background to make informed choices for them.

The facets that are discussed in this paper are *time-transitive flows*, *replaying flows*, *direct release* and *whitelisting flows*. We phrase each of these facets in terms of what it means to consider these flows as secure, and attempt to present the same flow in two contexts, each motivating whether the flow should be considered secure or not.

Note: It is important to point out that the (in)security of these examples is argued only using the code fragments and potential application context, not in the view of any particular security condition. This underlines the principle we aim to convey: when designing a security condition, first decide how the condition should treat these facets and then construct the right condition to match those choices, not vice-versa.

Remark 1 (Notation): In our examples we maintain the convention that program locations are labelled with a fixed security level. The lower-case first letter of the program location matches the upper-case first letter of the security level (e.g. `a` has level `A` and `hos` has level `Hospital`). We assume that initially no information flows are allowed between any two levels. The syntax `B` $\longrightarrow$ `A` changes this ordering and allows information to flow from level `B` to `A`, whereas `B` $\not\longrightarrow$ `A` revokes this permission. Note that these are only conventions used to present the flows; the flows themselves do not rely on them. This notation is taken from [5].

Time-transitive flows *A flow is time-transitive if it moves information from security level A to level C via a third level B, while there is no moment in time where the flow from A to C itself is allowed by the flow relation.*

Secure

```
  User ⟶ XSSFree
x := escapeHTML(uIn);
  User ↛ XSSFree
  XSSFree ⟶ DB
db := escapeSQL(x);
```

Insecure

```
  Patient ⟶ Hospital
hos := patData;
  Patient ↛ Hospital
  Hospital ⟶ DrPhil
drPhil := hos;
```

In a context where the flow of information reflects some declassifying or sanitising intention, one can argue that time-transitive flows are secure. In this example, user input `uIn` is first passed through the sanitiser `escapeHTML` to prevent XSS attacks, and later through the sanitiser `escapeSQL` before storing the information in the database. Here, it is of no relevance that the user input was never allowed to flow directly to the database.

The time-transitive flows facet was previously identified under the name *transitive flow* by Swamy et al. in the development of Rx [39]. We purposely refer to them as time-transitive flows to avoid confusion with intransitive non-interference [33], which is discussed further in Section IV-B. Swamy et al. argue that time-transitive flows should be considered insecure, using the following example. Patients allow their data to only flow to the doctors of the hospital while they are under treatment. When a patient leaves the hospital, this information should no longer be available and in particular not to doctor Phil, who joined the hospital staff after the patient has left. Here, it is sensible to disallow the time-transitive flow.

One way to differentiate between these in the security property is to either limit the attacker's increase in knowledge by what can be learnt from the observable part of the *current* memory (allowing time-transitive flows)[2] or from the *initial* memory (disallowing time-transitive flows).

Replaying flows *When the release of information is considered* permanent*, this information flow can be repeated without breaking the information security of the system.*

As an example, consider the scenario where the National Security Agency (NSA) releases a file to the US military. Once released, the military can access this file at any time, regardless of whether this information is currently in their possession.

---

[2] This can be translated into knowledge on the initial memories that could have resulted in the current observable memory.

Secure

```
  NSA ⟶ Military
mil := nsaFile
mil := 0;
  NSA ↛ Military
mil := nsaFile;
```

Insecure

```
  Creditcard ⟶ Log
log.write(cc);
log.clear();
  Creditcard ↛ Log
log.write(cc);
```

Considering information as permanently released is not the natural choice in every situation, as the insecure example demonstrates. Since the log file has been cleared, the credit card information is no longer available. Hence the effect of the earlier release has disappeared, and to store the same information in the log file again requires the flow relation to agree with this flow again.

To make this difference even more explicit, we include a second example for both the secure and insecure context which combines replay with time-transitive flows. If the NSA information has been permanently released to the military, when Bob later joins the military he should have access to this information as well, again regardless of whether it is currently in the military's possession. In the insecure context, the vendor is allowed to see the information in the log file, and since it does not contain the credit-card number, the vendor should not be allowed to observe it.

Secure

```
  NSA ⟶ Military
mil := nsaFile;
mil := 0;
  NSA ↛ Military
  Military ⟶ Bob
bob := nsaFile;
```

Insecure

```
  Creditcard ⟶ Log
log.write(cc);
log.clear();
  Creditcard ↛ Log
  Log ⟶ Vendor
vendor.receive(cc);
```

The restricting interpretation appears more natural when taking a language-based perspective on information release. The view of permanently releasing information matches more closely the original use of the term "declassification" in a military context, whereas the language-based approach is more related to how the same term "declassification" is often used in current information flow research. We return to this disambiguation of the term declassification in Section V.

The language-based view suggests that we can make a second distinction in this facet, which we call *weak replaying* of flows. Weak replay captures the idea that information is only considered released as long as it is still available at the level to which it was released. A motivating example for weak replays is again the setting where the same credit card information is added to the log file, but before the log has been cleared.

Secure

```
  Creditcard ⟶ Log
log.write(cc);
  Creditcard ↛ Log
log.write(cc);
```

Insecure

```
  Ezine ⟶ Customer
customer := ezine;
  Ezine ↛ Customer
customer := ezine;
```

To argue for the insecurity of weak replaying flows, consider a scenario where a customer pays for a time-limited subscription on an online magazine ("e-zine"). When the subscription runs out, the customer should no longer be able to download magazine, even if they have an old copy of the same edition.

This facet was previously identified by Askarov and Chong, and we use their approach as a technique for addressing this facet in its various degrees [5]. To allow for (strong) replaying of flows, we can set the attacker's observation power to remember all observations made. That is, $obs_A(t)$ could be said to be the sequence of events observable by A. Hence, after observing an event e which contains the same information flow in earlier observations $obs_A(t)$, we have that $ek_A(P, obs_A(t \cdot e)) = ek_A(P, obs_A(t))$ and the release is considered secure regardless of the current flow relation. To disallow any replaying of flows, we can consider attackers who do not have a perfect recall of all observations made, and to whom the replay may therefore come as a revelation.

Again, our aim is only to argue for the existence of facets, not to state that one treatment of a facet should be preferred over another. For replaying flows we see examples of both choices in the reviewed literature in Section III-C.

Although allowing for (only) weak replaying flows seems to arguably better match the language-based view, we are not aware of any literature that addresses the facet in exactly this way. One possible encoding would consider attackers without perfect recall, but allow the attacker to observe the (currently) non-secret part of the current memory.

Direct Release *A security condition supports direct release if information is considered released as soon as the current flow relation permits it to flow. This means that a revocation of that permission does not affect this information.*[3]

Secure

```
  Data ⟶ App
send(app, "Hello");
  Data ↛ App
send(app, data);
```

Insecure

```
  Salary ⟶ Screen
screen.show("Hello");
  Salary ↛ Screen
screen.show(salary);
```

Considering such flows secure can be justified if we model attackers as constantly observing, directly in memory, all the information which they have permission to know. As an example, the attacker could be an application running in parallel with the code displayed above. Hence it does not matter that no data was actively sent to the application, we consider the data as released directly when this is allowed. Note that direct release does not imply that revocation (changing the policy to be more restrictive) is irrelevant: new information that arrives at level `Data` (either via input channels or from a different security level) is considered to be not yet released to `App`.

On the other hand, we can argue that the same kind of flow is insecure when the attacker can only observe information that is actively provided. In the code above no information about the salary has been released to the screen, and hence it makes sense to assume that an observer does not know this information yet.

If we chose to allow direct release, we could reflect this in the security property by modelling an attacker's observation as the part of the current memory that the attacker is allowed to

---

[3] Note that despite the similarity in our examples, direct release is not merely an even stronger version of replaying flows. Direct release is concerned with what an attacker is assumed to have learned everything that is permitted at the point when the flow relation becomes more liberal – not whether the attacker may learn it again once the flow relation no longer explicitly permits it.

		T	R	D	W
NI	Swamy et al. (RX) [39]	+	-	+	+
	Hicks et al. [25]	+	-	-	+
BI	Boudol and Matos (Non-disclosure) [1]	+	-	-	+
	Broberg and Sands (Flow Locks) [11]	+	-	-	+
Epistemic	Askarov and Sabelfeld (Gradual Release) [4]	N/A	+	-	+
	Banerjee et al. (Flowspecs) [9]	N/A	+	-	+
	Balliu [7]	+/-	+	+/-	+
	Askarov and Chong [5]	-	+/-	-	+
	Broberg and Sands (Paralocks) [12,13]	+	+	-	+

T: time-transitive, R: replay, D: direct release, W: whitelisting

TABLE I. CLASSIFYING EXISTING SECURITY CONDITIONS ALONG THE FACETS. + INDICATES THAT FLOWS OF THIS FACET ARE ALLOWED, - THAT THEY ARE NOT. +/- SIGNIFIES THAT THE FACET IS NOT FIXED BY THE CONDITION. N/A DENOTES THAT THE FACET DOES NOT APPLY TO THIS SECURITY CONDITION. GROUPED BY NATURE OF CONDITION: NON-INTERFERENCE (NI), BISIMULATION (BI) OR EPISTEMIC.

observe according to the current flow relation. This opposed to only considering "active" flows, such as observing changes in the memory, which we could use if we want to consider direct release insecure.

Whitelisting flows *A security property is whitelisting if a flow is allowed whenever there is some part of the policy that allows for it.* This opposed to blacklisting, where a flow is disallowed whenever there is some part of the policy that does not allow it. The facet becomes apparent when a flow is permitted by one part of the policy, but denied by the other.

As an example of such a situation, and an argument for whitelisting, consider the release of an encryption key. It is reasonable to accept that with the release of this key an observer also learns the secret information that was earlier released encrypted under that key, even though part of the policy does not allow the secret to be released.

Secure

```
  Secret ⟶ Pub
  Key ⟶ Pub
output(k XOR secret);
  Secret ↛ Pub
output(k);
```

Insecure

```
  Bob ⟶ Report
  Carla ⟶ Report
r.avg := (b.s+c.s)/2;
  Carla ↛ Report
r.bob := b.s
```

The insecure example shows the creation of a report `r`, storing the average of the salaries `s` of Bob and Carla. Then, when Carla explicitly no longer allows information about her salary to flow to the report, we add Bob's salary to the report from which an observer can derive Carla's salary. This we could argue violates Carla's concern and should be regarded as an illegal flow.

Whitelisting appears to be the norm for language-based security conditions as is confirmed by the literature that we discuss in this paper, which all treat the policy as a whitelist of permitted flows. Treating the policy as a blacklist rather than a whitelist is more common in the interpretation of noninterference in event-based systems [22,45].

C. Classification of facets in literature

In this section we present a collection of existing security definitions for dynamic policies and classify them along our facets. Although we think that our discussion is rather complete, our goal is not to give a full survey of the field. Rather, the purposes of this section are *a*) to illustrate that the listed facets indeed occur in literature; *b*) to identify what components in a security condition determine which classification, as an aid for the developer of new security conditions; *c*) to demonstrate that the facets can be used as a terminology for discussing security conditions for dynamic policies (similar to the intention of Sabelfeld and Sands introducing the dimensions for declassification [35]); and *d*) to convince the reader that the intended facets should be considered before defining a security condition, to ensure that it matches these intentions.

With this setting in mind, we purposefully left out some literature that could be considered related to dynamic policies, but does not fit the goal of this section. Examples include the *non-interference until* conditions property from Chong and Myers [17] (since it is unspecific in its treatment of information once released) and λAIR by Swamy et al. [40] (which presents a type system for enforcing user-specified security properties, but does not present such properties itself).

An overview of the security conditions in the surveyed literature along our facets is listed in Table I. We remind the reader that the facets do *not* serve as an objective measure for the quality of a security condition: arguments can be made both in favour and against each facet as illustrated in Section III-B. Since none of the discussed literature interprets the policy as a blacklisting of information flows rather than a whitelisting, we omit this facet from this discussion. Similarly, direct release is not permitted by most properties by virtue of their observation model and we only discuss this facet for the properties that do permit it.

Remark 2 (Disclaimer): We present all conditions in the same computational model, despite some of the surveyed conditions being defined in a different context. As a consequence, some of the classifications in Table I do not directly apply on these original security properties. For example, time-transitive flows do not apply on a property for a model where observations are made on output channels instead of state changes. We allow ourselves to make the following transformations:

- *Modified sources*: We define sensitive information as the information in the initial state, instead of values input on channels.
- *Modified sinks*: We define observations as state modifications, instead of values output on channels.

We acknowledge that restricting ourselves to this single model may not be sufficiently general, and additional facets may be revealed under a different model (e.g. considering input channels instead of initial states, cf. Clark and Hunt [19]).

We here only discuss the conditions to the extent needed to understand their classification. An extended version of this section discussing several of the security properties in more detail can be found in the technical report version of this paper [15]. We group the security conditions according to whether they are of a non-interference, bisimulation, or epistemic nature.

1) Non-interference (NI) based conditions: These conditions are built directly on top of non-interference properties phrased in two-run style. Dynamic policies are enforced by varying how non-interference is included as a building block in the security condition, not by changing the non-interference property itself.

*a) Swamy et al. (*RX*) [39]:* With the clearly stated goal of disallowing time-transitive flows (referred to as *transitive flows*) RX introduces the notion of a *transaction*. Transactions are defined together with a fixed flow relation on parts of the information in the system. If this specified flow relation is modified during the transaction, the operational semantics rolls back to the pre-state of the transaction, undoing the changes to memory but preserving the changes in policy. This operational guarantee allows the static check to assume the fixed flow relation to hold throughout the transaction.

The security condition considers a program secure for attacker A if it is non-interfering between policy changes that are declassifying (i.e. that allow more flows to A). Therefore the security condition allows for the replay of flows only until the next declassifying policy change, after which the non-interference requirement is restarted, effectively "forgetting" the earlier flows and thus no longer allowing replays. We conclude that RX therefore does not (in general) support replaying of flows. Observations are modelled as projections of what the attacker may observe from the current memory under the current flow relation. Revocation thus does not affect the classification of released information, making the security condition permit direct release.

Interestingly, although disallowing time-transitive flows was the motivation behind the transaction system, the security condition itself does *not* rule them out. Consider the time-transitive flow of information from security level C to B, followed by B to A, as seen by an observer on level A. Since allowing the flow from B to A is regarded as a declassifying change for A, the non-interference condition restarts. Consequently the security condition only considers memory stores that agree on all values of level B for this second part of the flow, ruling the program as secure. This is as a clear example where the security property does not exhibit the intended facet.

b) Hicks et al. [25]: The Decentralized Label Model (DLM) is a well-established language for specifying information flow security labels based on principals [30]. The ordering between security labels is partly influenced by an *acts-for* hierarchy between these principals, which may vary over time. Jif, an extension to Java adding support for the DLM, assumes that these variations occur only very occasionally. This justifies branching on queries on the hierarchy, ruling the following program secure [29]:

```
if (A ⟶ B) { long_computation(); b := a; }
```

The work by Hicks et al. introduces a calculus that removes this assumption. Coercion checks (permission tags) are statically introduced to summarise the parts of the program for which constraints on the acts-for hierarchy need to hold in order to be secure. When, at run-time, an (asynchronous) request to change the acts-for hierarchy presents itself, this is delayed until it is consistent with the permission tags.

A program is said to be secure if it is non-interfering for an unknown, but fixed, acts-for hierarchy. Effectively, this results in the property *non-interference between policy updates*. To classify this condition along our facets, the flows need to be rephrased to use policy queries in a way that matches the appropriate policy changes. For example, a replay of flow can be exemplified by:

```
   A ⟶ B
if (A ⟶ B) { b := a }
   A ⟶̸ B
b := a
```

This example shows that the security condition does not allow replay, since this program is interfering for a fixed policy in which `A` may not flow to `B`. For a similarly phrased program with time-transitive flows there is no fixed policy for which the program is interfering, hence time-transitive flows are allowed.

2) Bisimulation (BI) based conditions: Bisimulations are used frequently to formulate security conditions. A style of bisimulation introduced by Sabelfeld and Sands [34], called *strong low-bisimulation*, arises in several works dealing with dynamic policies: the non-disclosure property by Boudol and Matos [1] as well as the original Flow Locks security condition by Broberg and Sands [11] in particular.

Strong low-bisimulation relates two programs P_1, P_2 if, when started in equivalent memory stores according to the current flow relation they perform observationally matching steps. In addition, the programs from the resulting configurations should also be bisimilar. A program is considered secure if it is bisimilar to itself. An important property of strong low-bisimulation is that it quantifies over all equivalent stores at each step. Effectively this means that it considers combinations of memory stores and commands which are impossible in a single-threaded system. For example, for the command `if (x > 0) { y := x}` bisimulation is required for the sub-command `y := x` for stores where $x \leq 0$.

Since bisimulation is required on all sub-commands regardless of previous flows, security conditions based on strong low-bisimulation do not allow for replaying of flows nor direct release. For the second half of a time-transitive flow, the condition considers only those memory stores equivalent by the current flow relation, thus allowing time-transitive flows.

3) Epistemic security conditions:

a) Askarov and Sabelfeld (Gradual Release) [4]: The use of epistemic formulations in security conditions for information flow was first found in the Gradual Release property. The policy language contains two security levels *Low* and *High*. The programming language includes a declassification primitive to allow information flows from *High* to *Low*, explicitly marking the resulting observations as "release events". The security condition is then that the knowledge of a *Low* observer should not increase on observations that are not release events. Here, the knowledge of an attacker includes both knowledge of the *Low*-labelled part of the initial store as well as what can be learned from the observations produced by the program.

Gradual Release is only defined for a setting with two security levels and there appears to be no natural extension for multiple levels without making determining choices on allowing transitive flows. We therefore classify this facet as not available for gradual release. Neither does the definition involve an actual dynamic policy as such. Hence in order to classify the definition along the facets they need to be phrased using the declassification primitive instead of policy change. For example, replaying flows can be exemplified with:

```
a := declassify(b)
a := b
```

As the first observation is a release event, the requirement that the observer's knowledge stays the same applies only on the second observation. Having already observed the value of `b` in the first observation, the second observation does not teach the observer anything new: replaying flows is allowed.

The concept of Gradual Release can be used to denote more fine-grained policies specifying *what* information may be revealed at each release event, see e.g. [6,8]. Such extensions do not change how the facets are addressed.

b) Banerjee et al. (Flowspecs) [9]: Banerjee et al. introduce a modular approach to statically enforcing secure information release. Standard type-checking for non-interference is employed on all program code except for those parts marked as declassifying. These declassifications need to agree with a *flowspec*, a specification of what information may be released and under which circumstances. Program verification is used to verify that each declassification matches a justifying flowspec.

The enforced security condition, *conditioned gradual release*, is very similar to gradual release except that for each releasing observation there has to be a flowspec which allows for this release. This additional expressiveness in the specification of policies has no impact on the classification in our facets, but can provide more guarantees than a gradual release policy. We discuss the expressiveness of policy specification mechanisms in detail in Section IV-C.

c) Broberg and Sands (Paralocks) [12,13]: The Flow Locks policy language and its successor Paralocks have similar security properties to which we refer collectively as Paralocks security. In the Paralocks policy language, information flows to actors are guarded by *locks* which can be opened and closed. That is, the flow relation depends on the set of open locks: the more locks are open, the more flows are permitted.

Based on gradual release, Paralocks security also defines attacker knowledge as the combination of what can be observed from the initial memory store and what is learned additionally from observations produced by the program. As identified for the previously discussed epistemic properties, this means that Paralocks security necessarily allows for replaying flows since this knowledge does not increase when the same information flows again.

An attacker consists of a pair of an actor (the observer) and a set of locks called *capabilities* (this attacker can observe as if these locks were open). Let $t \cdot e$ be a trace produced by some command c. The Paralocks security property says that, if the capability locks of the attacker were open at the moment of event e, the attacker's knowledge should remain the same.

This makes the property time-transitive. Let A be an attacker observing the second observation resulting from a time-transitive flow. If the locks necessary for this flow exceed A's capability, the security condition directly considers this secure for A. Otherwise, A would necessarily also have made the first observation of the time-transitive flow, making this effectively a replay for A which we already argued as secure.

d) Balliu [7]: In his work, Balliu shows how trace-based conditions such as separability and generalised non-interference fit in a generic security condition and how they can be characterised in epistemic temporal logic. This condition is of a format similar to the property discussed in Section III-A.

$$ek_A(P, obs_A(t \cdot e)) \setminus ek_A(P, obs_A(t)) \subseteq \mathcal{R}_{now}$$

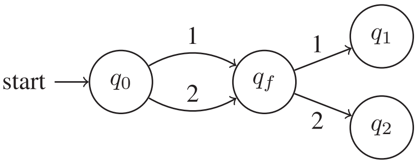

Fig. 1. An automaton modelling a forgetful attacker who only remembers the second value observed (in the domain $\{1, 2\}$), in the style of Askarov and Chong [5].

The security condition is parametric in the release policy $\mathcal{R}_{now}$, and therefore does not fix all of the facets until this release policy is instantiated. In the technical report version of this paper we show how this condition may both permit and deny direct release or time-transitive flows depending on the instantiation of the release policy [15].

The definition of knowledge *does* have a fixed definition: the attacker excludes any memory store that could not have led to the series of observations made. Once again, this implies that $ek_A(P, obs_A(t \cdot e)) \setminus ek_A(P, obs_A(t)) = \emptyset$ when event e is the replay of earlier flows, and replays are permitted.

e) Askarov and Chong [5]: The security condition by Askarov and Chong, by intention, does *not* necessarily allow for replaying information flows. As for Balliu, the security property of the form described in Section III-A, except that $\mathcal{R}_{now}$ is fixed to the set of stores that do not appear equivalent to the attacker A under the flow relation when the last event e was produced. As observed by Buiras and van Delft, this makes the security condition disallow time-transitive flows [16].

What sets this security condition apart is its definition of attacker knowledge. Rather than assuming an attacker with perfect recall, Askarov and Chong allow attackers to "forget" (parts of) earlier observations. An attacker A is modelled as a combination of a level and an automaton which makes transitions based on the observed values. As an example, the automaton in Figure 1 models an attacker who remembers the second output but forgets the value of the first observation.

Suppose that trace t puts an attacker's automaton in state s. Thus s models what the attacker can remember from trace t. The knowledge from a trace t is the set of states the attacker can exclude, namely those that could not have led the attacker in state s. Consider an attacker with the automaton from Figure 1 observing the output on level A of the following program (assuming that $\texttt{b} \in \{1, 2\}$):

```
  A ⟶ B
b := a
  A ↛ B
b := a
```

Only after the second observation does this attacker learn the value of `b`, that is at a time when this is *not* allowed by the current flow relation. Hence, the forgetful attacker model is an effective way to rule out replays of previous information flows. Replays can still be permitted if only the perfect-recall attacker is considered.

Unfortunately, as identified by Askarov and Chong, demanding security against *all* possible forgetful attackers could be argued unreasonable as it includes "wilfully stupid" and unrealistic attackers. Van Delft et al. [42] identify a definitive set of attackers for a progress-insensitive version of the security condition, but identifying a reasonable set for the progress-sensitive version remains an open question.

D. Directions for Future Research

It could be useful to formulate a generic security property which is modular in its facet classification. Such a property would allow the designer to make a choice for each facet, and then simply achieve this by enabling or disabling the necessary components in the generic property (although not all combination of facets may be possible, as the existing properties in Table I suggests). The security property by Balliu exhibits some of this genericity, but lacks identifiable components for each facet.

We consider it possible that our set of facets is incomplete and additional facets of security properties will be identified in the future. For each new facet, we recommend a similar treatment to survey existing security conditions and identify what influences its classification.

When the security condition determines the facets, all information releases are treated equally. An interesting generalisation would be to let the program explicitly state the *intended facets per release*. For example, a program could annotate some releases as "permanent", indicating to the security property that these may be replayed (as opposed to the unannotated releases). The security property has to treat such specified intentions properly.[4]

IV. CLASSIFICATIONS OF DYNAMIC POLICIES

In this section we take a closer look at the anatomy of policy schemes and policy specification mechanisms, with the intention of introducing a standard terminology for discussing and comparing them. We discuss the different *control levels* at which a scheme can operate, and how these levels taken together, in combination with assignment of labels in a system, form what we generally refer to as a policy. We further use this anatomy as an overlay to shed new light on the "dimensions of declassification" introduced by Sabelfeld and Sands [36], and resolve some perceived unclarities and ambiguities among the dimensions. We categorise existing policy specification mechanisms in the literature accordingly. Finally we derive a framework for reasoning about the kinds of *invariants* a mechanism can express in a generated policy scheme, as a starting point for comparing expressiveness between different mechanisms.

A. The anatomy of dynamic policies

A dynamic policy scheme can be understood, discussed and classified in terms of a *hierarchy of control*, consisting of the following control levels:

- Level 0 control – $\mathcal{F}$, a *set* of possible *flow relations* between the information sources and sinks available in the system.
- Level 1 control – δ, a *determining function* selecting which flow relation in $\mathcal{F}$ should be active. We refer to the arguments to this function as the *discriminator*.
- Level 2 control – μ, a *meta policy* controlling the way in which the current flow relation may be changed.

The control levels allow us to be explicit about what it is that makes a policy dynamic: the possibility to define a determining function *and* have an input to this function that changes during program execution. Arguably, one could have a meta-meta policy which in turn controls the meta policy. However, with no loss of generality we group all further abstractions in the *meta policy* class, since the meta policy can also be taken to control itself.

As a concrete example, consider a scheme consisting of two security labels, *Secret* and *Public*. Potentially the set $\mathcal{F}$ could contain all four possible flow relations that involve these two security labels, but in this example it contains only two: one in which *Public* information may flow to *Secret* but not vice-versa; and one in which information may also flow from *Secret* to *Public*. The former flow relation is the default, and the latter is available when using a special "declassify" operator. The determining function δ then decides whether information may be released from *Secret* to *Public* or not, based on the current value of its argument, S. In this example, S can range over boolean values, indicating whether "declassify" is used or not. The meta-policy simply always allows transitions back and forth between true and false.

Formally we characterise the determining function as:

$$\delta : S \to \mathcal{F}$$

Here S is the information used to determine which flow relation is currently active. S could range over boolean values as in our simple example; it could be partial information from the current program state, as is the case in Paragon [14] or the work by Chong and Myers [17]. S may also consist of other information such as lexical location in the source code (as is done by Boudol and Matos [1]) or 'asynchronous' information external to the program (e.g. the acts-for hierarchy among principals used by Hicks et al. [25]).

In turn, the meta policy controls the changes between flow relations caused by the determining function. Strictly speaking, a meta policy aims to control the transitions between flow relations, and does not require the determining function as a "proxy". For example, a meta policy could, again based on some (meta) information M such as program state, impose a constraint on the progress of flow relations:

$$\mu : M \to 2^{\mathcal{F} \times \mathcal{F}}$$

Here the pair $(f_1, f_2) \in \mathcal{F} \times \mathcal{F}$ indicates that transition from flow relation f_1 to f_2 is permitted. In this way, it would be easy to for example define a μ specifying the meta policy that the flow relation between information only becomes more liberal. However in the surveyed literature, we find that the meta policy is typically defined in terms of controlling how the information S used by δ may change, rather than the resulting flow relation determined by δ – simply an extra level of indirection. That is, a meta policy instead typically has the characterisation:

$$\mu : M \to 2^{S \times S}$$

---

[4] One could argue that RX already presents such a system. By placing code in a transaction the programmer indicates the intention to disallow time-transitive flows within this code block. However, as argued in the discussion of RX, these intentions are ignored by the security condition which still treats all releases as allowing for time-transitive flows.

Note that the determining function does not solely serve as a level of indirection for the meta policy: the meta policy specifies how the current flow relation *may* be changed, but it is the determining function that specifies what the current flow relation *is*.

To demonstrate that the control hierarchy functions as a terminology for policy specification mechanisms, we show how various mechanisms from literature fit onto these levels:

- The programming language Jif [29] has a declassification function which can be described as temporarily changing S, making the intent clear such that δ provides a flow relation which allows for the declassification. The relation between declassification and dynamic policies is discussed in more detail in Section V. The decision to declassify is restricted by components such as authority and declassification robustness [44], which constitute M. Both δ and μ are pre-determined. The flow relations in $\mathcal{F}$ are determined by the labels used, in combination with the *acts-for* hierarchy. If changes to the acts-for hierarchy are allowed, as per Hicks et al [25], then this hierarchy is also included in S.
- The programming language Paragon [14] allows for the specification of Paralocks policies (see Section III-C) in a Java-like language. Hence the current flow relation is determined by the lock state, which constitutes S. In turn, these locks are information in the program state and are protected by locks themselves[5], making that second set of locks determine the meta-policy, i.e. when the first set of locks can be opened and closed. The labels include specification of how they are interpreted with respect to changes in the lock state, meaning that the set of flow relations, and the behaviour of δ (as well as μ) can be customised for each specific policy scheme.
- The programming language Rx [39] incorporates the *RT* framework [26] to specify a flow relation among *roles*. The active flow relation is determined by the set of memberships and delegations specified on each role, which form S. Roles also carry labels which specify who can observe the current members of a role, hence forming the set M determining on which secret data the decision to change flow relation (add or remove memberships and delegations) may depend.
- Matos and Cederquist [2] present a security condition for distributed computations (Distributed Non-interference). The default flow relation between security labels can be relaxed using the lexical construct from earlier work on the non-disclosure property [1], making S the locality in the code. These scopes with more liberal flow relations may only be entered when the node on which the computation runs allows for it. Each node specifies its own regulation on the allowed added flows, making M the locality in the network.

We note that the apparently clear separation offered by the levels of control is not necessarily mirrored in actual specification mechanisms. As noted, in Paragon each data source and sink is annotated with a security label that specifies not only a static behaviour, but also how that label should be interpreted in different lock states. In other words, the information on what the flow relations are and how they are determined is distributed across all the labels in a program, not cleanly as a single determining function. This does not mean that Paragon could not still be understood and described in terms of the levels of control.

	What	Who	Where	When
$\mathcal{F}$	+	-	Level locality only	-
δ	-	+	+	+
μ	-	+	+	+

TABLE II. Revisiting the declassification dimensions; + indicates that a dimension concern can be addressed by that policy component, - that it cannot.

Another observation is that the higher levels of control, δ and μ, have two components where control can be exercised: in the definition of δ, resp. μ, and in the argument to the respective function. To contrast these two possibilities, Rx allows control over the argument provided to μ (who can observe the members of a role), but μ itself is fixed. This is opposed to the meta-policy by Matos and Cederquist where the argument to μ is fixed to be the node on which the computation runs, but μ itself can be defined by the policy designer.

B. Rethinking the Dimensions of Declassification

Looking at the literature, it is clear that the four "dimensions of declassification" introduced by Sabelfeld and Sands [36] – "what", "who" "where" and "when" – are significantly different from each other in nature. In particular aspects of the "what" dimension are largely orthogonal from the other three, while many uses of "where" and "when" often coincide. There are also different aspects grouped within the same dimension that are so disparate as to be incomparable. For example, the "where" dimension intends to cover both *code locality* and *level locality*, which are only remotely related at best.

We propose that these dimensions should be discussed for each of our identified dynamic policy levels *individually*. We identify that not every dimension is relevant for every policy level, as summarised in Table II. This insight resolves some of the confusion in the declassification dimensions, showing that by taking the anatomy of a policy into account while discussing its "declassification' dimensions, we arrive at a clearer framework for discussing and comparing security conditions.

We present a short summary of each of the declassification dimensions and discuss them with respect to the policy anatomy.

What – Policies can dictate that only parts of a secret may be released (e.g. the last digits of a credit card). In addition this dimension covers quantitative release which is better characterised by "how much" and can be achieved using an information-theoretic approach (e.g. [18]). Although the decision to e.g. increase the amount of information that may be released comes from different components, the possibility to *express* this dimension only exists naturally in the ordering between information itself, i.e. when specifying flow relations.

Who – This dimension is concerned with being able to express who controls the release. In particular, it is sensible to prevent

---

[5] We observe that the possibility to place locks on policies is part of the specification mechanism used in Paragon, but not included in the Paralocks specification language.

an attacker from abusing the release mechanism, as is the motivation for robust declassification [44]. Since the decision to declassify can be controlled both by the determining function and the meta policy, this dimension can be addressed on either level. By nature this dimension talks about control over flow relations, and therefore is not relevant on the level of the flow relation itself.

Where – This dimension is split into two different forms of locality: *level* locality and *code* locality.

Level locality addresses the concern where information may flow relative to the security levels of the system. This dimension is particularly present in *intransitive non-interference* [33], which is exemplified by a policy which allows information to flow from security level *Secret* to *Declassify* and from *Declassify* to *Public*, but not directly from *Secret* to *Public*. This can be expressed in a flow relation using downgrading relations (Mantel [27]), but could also be addressed by the δ and μ controls on the ordering. The latter is achieved by changing between flow relations such that only one of the two flows is permitted at any specific time (this essentially requires *time-transitive flows*, see Section III-B).

Code locality allows policies to describe where syntactically in the code information may be released. One could sort of view this as level locality except the information should not pass through the *Declassify* level but through a lexical declassification construct in the program's code. Similar for the Who dimension, code locality is concerned with controlling which flow relation is active, and can therefore only be addressed by δ or μ.

When – A policy can dictate that information may only be released after (or before) a certain time has passed. This temporal restriction can be based on various elements, such as real time, the size of the secret or relative to other events in the system. Although the original presentation of this dimension splits it into various classes, all temporal controls need to be addressed by δ and μ as they concern the decision to change the ordering of information.

When we now reclassify policy mechanisms by the levels first and the dimensions second, the classification becomes clear and unambiguous. We briefly show this for the examples used in Section IV-A. [6]

For Jif, the discriminator for the determining function δ is given by a combination of the declassification construct, which concerns the "where" dimension (both code and level locality), and the acts-for hierarchy, which concerns the "who" dimension. As a meta-policy, authority provides a meta control on the decision to declassify in the "who" dimension. Robustness does so as well, and in addition partially addresses the "what" dimension by limiting what information can be declassified. For Paragon, both δ and μ are regulated by locks which concern the "when" dimension: information flows are allowed relative to the actions of opening and closing locks. Paragon also has a lexically scoped version of opening a lock, which works in the "where" dimension (code locality). Implementations in Jif and Paragon can combine the programming and policy language to encode requirements in other dimensions [3,14,41], but these are not a natural part of the policy language.

Both δ and μ in Rx use the "who" dimension: who is a member of a role determines the active flow relation, and who can view the current members of a role decides what policy change can be made. The security framework for distributed non-interference by Matos and Cederquist finds a fit in the "where" dimension for δ as the lexical flow construct determines where in the code the additional flows are allowed. The meta policy also fits in that dimension, as it determines where in the network each flow construct is allowed.

C. *The expressiveness of policy languages*

Different policy specification mechanisms offer a variety of expressiveness, from the simplest fixed two-level systems, up to full policy specification *languages* like those found in Jif [31], Rx [39] or Paragon [14]. We can have intuitive ideas regarding the relative expressiveness of such mechanisms, but what are the measures by which we can compare them formally? In this section we speculate on how a formal framework for comparison could be constructed.

Montagu et al [28] construct a framework for comparing "label models", or *policy schemes* in our terminology. We argue that such an approach is too simple for comparing expressiveness of full policy specification languages.

Consider a policy scheme which has three labels called TopSecret, Secret and Public. By default, the flow relation consists of the three flows Public → Secret, Public → TopSecret and Secret → TopSecret, so the labels form a strict hierarchy of security levels. The policy scheme also allows data to be declassified from Secret to Public. When declassifying, the flow relation is then the same three flows from before, with Secret → Public added. These are the only two flow relations possible. We refer to this scheme as TSP.

A second scheme has three labels simply called A, B and C. Any of the six possible flows between two labels can be allowed or not independently of other flows. In other words, all 2^6 conceivable flow relations involving these three labels are possible, and a programmer can freely change between them. We refer to this scheme as ABC.

A first attempt at comparing expressiveness could look at the possibility to *embed* one *policy scheme* in the other[7]. That is, the embedding scheme should contain at least the same set of flow relations as the embedded scheme. In this case we could embed TSP in ABC using TopSecret $= A$, Secret $= B$ and Public $= A$, and use the corresponding two matching flow relations. We could then claim that the second is at least as expressive as the first, by virtue of having at least as many labels allowing at least the same flow relations. However, such an attempt misses an important aspect of expressiveness. If we were to use ABC in place of TSP, what (other than regimen) stops us from making one of the "other" flow relations active? In particular, we have no guarantees that we will not use a flow relation in which A, proxying as TopSecret, can flow to other labels. ABC is certainly more *flexible* than TSP – but when

---

[6] None of the considered examples addresses the "what" dimension, or support intransitive flows in the flow relations, thus we do not discuss this dimension further.

[7] This is the comparison done by Montagu et al [28].

embedding, added flexibility is not a good thing. Restrictions matter!

For a policy scheme, the degree of flexibility is already fixed, so there is never any room for expressiveness. Truly, expressiveness should be compared at the level of policy specification mechanisms. Consider mechanisms PSM_1 and PSM_2: we have that PSM_1 is at least as expressive as PSM_2 if, for every possible policy scheme that PSM_2 can generate, PSM_1 can generate a scheme that can embed it – *including restrictions*. But how can we express restrictions formally?

The examples used so far show the need for restrictions at the level of what flow relations are possible. However, not all such restrictions are equally important. For our example above, the fact that when using ABC we could end up in contexts where the flow relation allows *fewer* flows than any of the ones matching those of TSP, is arguably acceptable – the system would still be secure. But the fact that we could end up in a context where A, representing TopSecret, can flow at all is not acceptable, as it means that using ABC we cannot give the same security guarantees *by construction*. Hence, what matters is the ability to express *invariants* over flow relations – specifically, invariants that concern the *absence* of some (set of) flows.

In this work we identify two principal forms of invariants that we want the ability to express. The first form are the *invariants over sets of flow relations*. Such invariants can be global, for example "no flow from TopSecret to any other level is ever allowed"; or *conditional*, for example "flows from Secret to Public are not allowed, except when declassifying". The second principal form are the *invariants over sequences of flow relations*. A simple example could be the Gradual Release property [4] that the policy may only change to become more liberal over time. Another more complicated example is a strong Chinese Wall property stating that if a flow CompanyOne $\rightarrow X$ has ever been allowed at any point, then $X \rightarrow$ CompanyTwo may not be allowed at any point in the future [10].

To formalise these notions we first observe that given some starting state S_0 and the set of possible transitions as given by the range of μ[8], we can enumerate all possible sequences of discriminators by iteratively applying all possible transitions. Let $\overrightarrow{S}$ be the set of all such sequences. A global invariant over sets of reachable flow relations is then a property Φ such that

$$\forall S_0 \cdot \ldots \cdot S_n \in \overrightarrow{S}.\Phi(\delta(S_0)) \wedge \ldots \wedge \Phi(\delta(S_n))$$

holds. A *conditional* invariant adds a filter Ψ such that

$$\forall S_0 \cdot \ldots \cdot S_n \in \overrightarrow{S}. \\ [\Psi(S_0) \Rightarrow \Phi(\delta(S_0))] \wedge \ldots \wedge [\Psi(S_n) \Rightarrow \Phi(\delta(S_n))]$$

holds. An invariant over *sequences* of flow relations is a property Φ such that

$$\forall S_0 \cdot \ldots \cdot S_n \in \overrightarrow{S}.\Phi(\delta(S_0) \cdot \ldots \cdot \delta(S_n))$$

holds. We can easily imagine a conditional version of invariants over sequences too, with a similar filter based on the domain of μ, however we have not identified any compelling examples.

Comparing expressiveness of policy languages

A policy specification mechanism PSM_1 is at least as restrictive as another mechanism PSM_2, if for every possible policy scheme that PSM_2 can generate, PSM_1 can generate a scheme that can embed it, including enforcing the same (negative) invariants.

The kinds of invariants we have categorised here capture the majority of all conceivable invariants that we may want a policy scheme to enforce, however, there are more complex invariants that cannot be expressed in these terms. Our invariants are essentially *safety properties*, and not all desired invariants can be expressed in terms of these. Our framework of invariants should thus be seen as a starting point for formalising comparisons between policy specification mechanisms, not a completed journey, and the formalisation of further points in the space of invariants is an open research question.

V. RELATION TO DECLASSIFICATION

The term *declassification* (or more generally *downgrading*) has long been used to signify the deliberate change of security label on data, to allow it to be used more liberally than before. This is not specific to research on information flow, or even security in general – the term is used with this meaning outside of technical contexts as well.

For information flow specifically, however, the exact meaning of the term is not clear. It has been used, we argue, to refer also to things that are not aligned with the natural definition given above.

We will first argue for proper uses of the term declassification, and consequently also for what we consider mis-uses of the term. We will then go on to discuss, within the proper uses, different meanings that can be given to the term; specifically we identify two different *flavours* of declassification, discuss their distinctive differences, and relate these to the facets from section III-B.

A. The term "declassification"

For terminology, we want to establish a clear difference between the use of the term "declassification" (or "downgrading") and dynamic policy change. In technical terms, mechanisms that achieve declassification can be treated as specific uses of dynamic policies, but the concepts are not equivalent. The important distinction we want to make is that declassification is, by nature, *data-centric*, as opposed to affecting a policy. It is possible to declassify a piece of data, but it is *not* possible to "declassify" a policy. A policy can be changed to become more (or less) liberal – data can be declassified and then *used* more liberally. Further, it seldom makes sense to talk about "declassifying" a program or a

---

[8] If we also know how the argument M to μ may change over time, we can have better precision than considering the whole range of μ. As argued, this could be accomplished using yet another level of meta-policy that governs how M may change, or baking this into μ itself.

computation[9] – instead that program or computation can be said to run *under a more liberal policy*.

Historically this distinction has not been entirely clear, leading to some confusion about specifically which aspects of a policy specification mechanism or semantic property that are referred to when talking about the declassification taking place in a system. In particular the survey on "Dimensions and principles of declassification" by Sabelfeld and Sands [36] uses the term "declassification" as an umbrella to include everything related to dynamic policies – and even some mechanisms which are purely static orderings. We now hope to foster a more fine-grained use of the terminology.

B. Flavours of declassification

In the literature we identify two distinctly different flavours of the (proper) use of the term: *relabelling* and *copying release*.

Relabelling In classic military terms, declassification corresponds to the physical operation of changing the security classification of a document. This process is mirrored in some systems that declassify data by effectively replacing its label with a more liberal one. We refer to this as the *relabelling* approach to declassification. In information flow systems, this behaviour can be simulated by changing a sufficiently fine-grained global policy in such a way that it puts less restriction on the usage of the data in question. Gradual Release [4] is an example of such a policy. It should be clear that this use of the term can be seen as a direct instance of dynamic policies, where the policy is made successively more liberal.

Copying Release The other flavour of the term declassification is the systems that use a specific declassifying *operator*[10] that allows a single exceptional flow that would otherwise violate the prevailing policy. In effect, declassifying in this sense creates a *copy* of the original data[11] available under a more liberal label. We refer to this as the *copying release* approach. Systems with this form of declassification include Jif [31] and JOANA [37]. Semantic properties for such systems are typically phrased in terms of a flow against the normal ordering being allowed specifically if the operation causing it is a distinguished declassification operation.

We can also view such operators as instances of dynamic policy change, where an application of the operator corresponds to a sequence of operations in which: the global policy is temporarily weakened, the flow happens, and the policy is restored to its previous state. This is how declassification is typically encoded in e.g. Paragon [14]. Note that this works in a sequential setting; when we introduce concurrency one would need to prevent concurrent threads from exploiting the temporary policy change, for example by making the sequence of operations atomic.

Interestingly, a semantic property intended to accommodate such operations is restricted in the choices that can be made for the various facets introduced in Section III-B. Table III summarises the compatibility of the facits with the two flavours of declassification.

	T	R	D	W
Relabelling	+	+	+/-	+
Copying release	+	-	+/-	+

T: time-transitive, R: replay, D: direct release, W: whitelisting

TABLE III. NECESSARY FACETS FOR A SEMANTIC PROPERTY ACCOMMODATING DIFFERENT DECLASSIFICATION FLAVOURS.

The information made available by a copying release is intended to be persistent, but not permanent. The operator effectively releases a *copy* of the information to a location with a different security level. This makes the release persistent, since the information can now be accessed from that security level instead, and the original classification of the copy is forgotten. As a consequence an accompanying security property needs to allow for both time-transitive flows and whitelisting. The release is, however, not permanent, as the released information is intended to be accessed *only* from the location containing the copy. If the declassified copy of the data is deleted, it is simply no longer available under the liberal label. This demands that the security property does not allow (weak) replaying of flows.

The only facet (of the ones we discuss in this paper) that is not fixed when employing a declassification operator, is in the treatment of direct release. Direct release can be allowed *only* if the policy is fine-grained enough to distinguish between each possible data item to be declassified. In practice, if the result of arbitrary expressions can be declassified, direct release would not be feasible.

VI. CONCLUDING DISCUSSION

We reiterate that our aim has been to synthesise knowledge about dynamic policies, with the purpose to increase understanding and help facilitate future work within the domain of information flow control. Our anatomy of policy schemes can give would-be authors of policy specification mechanisms a better understanding of the nature of what they propose, and the tools to sharpen it to achieve the desired expressiveness. Would-be creators of programming languages and systems incorporating information flow policies can draw inspiration and understanding from our discussion on the nature of declassification, and further look to our facets to make conscious choices regarding the nature of their security properties, to ensure that they truly capture the desired degree of security. And interested researchers can draw inspiration from the less illuminated and understood corners and areas that we leave uncovered or identify as open research questions. In short, we believe that the foundations laid down in this paper will make future work on information flow control sharper, easier, and stronger.

Acknowledgments This paper benefited from the comments of Musard Balliu, Pablo Buiras, Owen Arden, Sebastian Hunt, Andrei Sabelfeld, and the anonymous reviewers. This work is partly funded by the Swedish funding agencies SSF and VR.

REFERENCES

[1] A. Almeida Matos and G. Boudol, "On declassification and the non-disclosure policy," in *Proc. IEEE Computer Security Foundations Workshop*, 2005, pp. 226–240.

---

[9] Unless one is literally revealing a previously secret piece of source code.

[10] More generally *operation* – it does not need to be an operator per se, even if that is the most common case.

[11] More generally, the result of an expression whose result depends on the original data.

[2] A. Almeida Matos and J. Cederquist, "Distributed Noninterference," in *Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing*. IEEE Computer Society, 2014, pp. 760–764.

[3] A. Askarov and A. Sabelfeld, "Security-typed languages for implementation of cryptographic protocols: A case study," in *Proc. European Symp. on Research in Computer Security*, ser. LNCS, vol. 3679. Springer-Verlag, 2005.

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

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

[6] A. Askarov and A. Sabelfeld, "Tight Enforcement of Information-Release Policies for Dynamic Languages," in *IEEE Computer Security Foundations Symposium*, 2009, pp. 43–59.

[7] M. Balliu, "A logic for information flow analysis of distributed programs," in *Secure IT Systems*. Springer Berlin Heidelberg, 2013, vol. 8208, pp. 84–99.

[8] M. Balliu, M. Dam, and G. Le Guernic, "Epistemic temporal logic for information flow security," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '11. ACM, 2011, pp. 6:1–6:12.

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

[10] D. F. Brewer and M. J. Nash, "The chinese wall security policy," in *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*. IEEE, 1989, pp. 206–214.

[11] N. Broberg and D. Sands, "Flow locks: Towards a core calculus for dynamic flow policies," in *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, ser. LNCS, vol. 3924. Springer Verlag, 2006.

[12] N. Broberg and D. Sands, "Flow-sensitive semantics for dynamic information flow policies," in *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS 2009)*. Dublin: ACM, June 15 2009.

[13] N. Broberg and D. Sands, "Paralocks – role-based information flow control and beyond," in *Symposium on Principles of Programming Languages (POPL)*. ACM, 2010.

[14] N. Broberg, B. van Delft, and D. Sands, "Paragon for Practical Programming with Information-Flow Control," in *Programming Languages and Systems*, ser. LNCS, 2013, vol. 8301, pp. 217–232.

[15] N. Broberg, B. van Delft, and D. Sands. (2015) The Anatomy and Facets of Dynamic Policies - Technical Report. arXiv:1505.02021 [cs.CR]. http://arxiv.org/abs/1505.02021.

[16] P. Buiras and B. van Delft. Dynamic Enforcement of Dynamic Policies - Technical Report. http://slio.bitbucket.org/slio-tr.pdf. Accessed: 2015-02-11.

[17] S. Chong and A. C. Myers, "Security policies for downgrading," in *ACM Conference on Computer and Communications Security*, Oct. 2004, pp. 198–209.

[18] D. Clark, S. Hunt, and P. Malacaria, "Quantitative analysis of the leakage of confidential data," in *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, ser. ENTCS, vol. 59. Elsevier, 2002.

[19] D. Clark and S. Hunt, "Non-Interference for Deterministic Interactive Programs," in *Proc. 5th International Workshop on Formal Aspects in Security and Trust (FAST2008)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2008.

[20] E. S. Cohen, "Information transmission in computational systems," *ACM SIGOPS Operating Systems Review*, vol. 11, no. 5, pp. 133–139, 1977.

[21] R. Focardi and R. Gorrieri, "A classification of security properties for process algebras," *J. Computer Security*, vol. 3, no. 1, pp. 5–33, 1995.

[22] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. on Security and Privacy*, Apr. 1982, pp. 11–20.

[23] J. Y. Halpern and K. R. O'Neill, "Secrecy in multiagent systems," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 1, pp. 5:1–5:47, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1410234.1410239

[24] B. Hicks, K. Ahmadizadeh, and P. D. McDaniel, "From languages to systems: Understanding practical application development in security-typed languages," in *ACSAC*. IEEE Computer Society, 2006.

[25] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies," in *Foundations of Computer Security Workshop*, Jun. 2005, pp. 7–18.

[26] N. Li, J. Mitchell, and W. Winsborough, "Design of a role-based trust-management framework," in *IEEE Symposium on Security and Privacy*, 2002, pp. 114–130.

[27] H. Mantel, "Information flow control and applications—Bridging a gap," in *Proc. Formal Methods Europe*, ser. LNCS, vol. 2021. Springer-Verlag, Mar. 2001, pp. 153–172.

[28] B. Montagu, B. C. Pierce, and R. Pollack, "A theory of information-flow labels," in *Proceedings of the 2013 IEEE Computer Security Foundations Symposium*, Jun. 2013.

[29] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1999, pp. 228–241.

[30] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 410–442, 2000.

[31] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," Jul. 2001–2006, software release. http://www.cs.cornell.edu/jif.

[32] S. Preibusch, "Information flow control for static enforcement of user-defined privacy policies," in *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 133–136.

[33] J. M. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep. CSL-92-02, 1992.

[34] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proc. IEEE Computer Security Foundations Workshop*, Jul. 2000, pp. 200–214.

[35] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2005, pp. 255–269.

[36] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 15, no. 5, pp. 517–548, 2009.

[37] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab, "Checking probabilistic noninterference using joana," *it - Information Technology*, vol. 56, pp. 280–287, Nov. 2014.

[38] A. Stoughton, A. Johnson, S. Beller, K. Chadha, D. Chen, K. Foner, and M. Zhivich, "You sank my battleship!: A case study in secure programming," in *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, ser. PLAS'14. ACM, 2014, pp. 2:2–2:14.

[39] 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.

[40] N. Swamy and M. Hicks, "Verified Enforcement of Stateful Information Release Policies," in *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '08. ACM, 2008, pp. 21–32.

[41] B. van Delft, N. Broberg, and D. Sands, "Programming with Paragon," in *Proc. 2013 Marktoberdorf Summer School*, ser. NATO Science Series, 2013.

[42] B. van Delft, S. Hunt, and D. Sands, "Very Static Enforcement of Dynamic Policies," in *Principles of Security and Trust*. Springer, 2015.

[43] A. Zakinthinos and E. Lee, "A general theory of security properties," in *In Proceedings of the IEEE Symposium on Security and Privacy*. Society Press, 1997, pp. 94–102.

[44] S. Zdancewic and A. C. Myers, "Robust declassification," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2001, pp. 15–23.

[45] C. Zhang, "Conditional Information Flow Policies and Unwinding Relations," in *Trustworthy Global Computing*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7173, pp. 227–241.

[46] L. Zheng and A. C. Myers, "Dynamic security labels and noninterference," in *Formal Aspects in Security and Trust*. Springer, 2005, pp. 27–40.

Appendix A

Glossary of terminology

Copying release

Making a copy of (derived) data available under a more liberal *security label*.

Declassification

The deliberate change of the *security level* on data to allow it to be used more liberally.

Determining function

Function that, based on its arguments (the *discriminator*), determines a *flow relation*.

Dimension (of declassification)

A classifying axis on the basis of the *declassification* goal (what, where, when, or to whom information may flow).

Direct release

A *Facet*; information is considered released as soon as the current flow relation allows it to flow.

Discriminator

Argument to the *determining function*.

Downgrading

See *Declassification*.

Dynamic policy

Information flow policy under which the *flow relations* may change during computation.

Facet

An aspect of a *security condition* that determines whether a particular class of information flows is accepted as secure.

Flow relation

Relates components in the program or system between which information is permitted to flow, e.g. as an ordering between *security labels*.

Exclusion Knowledge

The set of secrets that could not have produced a given observation.

Hierarchy of control

Division of a *policy scheme* in three levels of control, each level controlling the one below it.

Information flow policy

Specification of the information flows permitted during program execution.

Knowledge

The set of all secrets that could have produced a given observation.

Meta policy

Specification of the way in which the current *flow relation* may be changed.

Noninterference

Security condition that defines absence of information flow, typically by saying that changing a secret input will not cause a change in public outputs.

Policy

See *information flow policy*.

Policy scheme

A set of *flow relations* and transition between them, in isolation from any particular program or system.

Policy specification mechanism

Mechanism or language to construct a *policy scheme*.

Relabelling

Replacing the *security label* on information, placing less restrictions on the usage of that information.

Release Policy

Determines what information may be released with each observation, possibly based on various aspects such as the current flow relation when the observation was produced, the attacker, or the produced observation.

Replaying flows

A *Facet*; information that has flowed previously can flow again, regardless of what the flow relation dictates.

Security condition

Semantic specification of when a program satisfies a given security policy.

Security label

A label attached to particular parts of the program or system in order to express security concerns.

Static policy

Information flow policy under which the *flow relation* may not change during computation.

Termination insensitivity

A *Facet*; when information that is revealed by the termination or output progress of an application is always permitted by the *security condition*.

Time-transitive flows

A *Facet*; information from *security level* A may flow to level C via a third level B, while there is no moment in time where the *flow relation* allows flows from A to C directly.

Whitelisting flows

A *Facet*; an information flow is allowed whenever some part of the *flow relation* permits it.