Expressing Information Flow Properties

Elisavet Kozyri[1], Stephen Chong[2] and Andrew C. Myers[3]

[1] *UiT The Arctic University of Norway, Norway; elisavet.kozyri@uit.no*
[2] *Harvard University, USA; chong@seas.harvard.edu*
[3] *Cornell University, USA; andru@cs.cornell.edu*

ABSTRACT

Industries and governments are increasingly compelled by regulations and public pressure to handle sensitive information responsibly. Regulatory requirements and user expectations may be complex and have subtle implications for the use of data. *Information flow properties* can express complex restrictions on data usage by specifying how sensitive data (and data derived from sensitive data) may flow throughout computation. Controlling these flows of information according to the appropriate specification can prevent both leakage of confidential information to adversaries and corruption of critical data by adversaries. There is a rich literature expressing information flow properties to describe the complex restrictions on data usage required by today's digital society. This monograph summarizes how the expressiveness of information flow properties has evolved over the last four decades to handle different threat models, computational models, and conditions that determine whether flows are allowed. In addition to highlighting the significant advances of this area, we identify some remaining problems worthy of further investigation.

Elisavet Kozyri, Stephen Chong and Andrew C. Myers (2022), "Expressing Information Flow Properties", Foundations and Trends® in Privacy and Security: Vol. 3, No. 1, pp 1–102. DOI: 10.1561/3300000008.

1

Introduction

1.1 Information Flow Properties for Today's Digital Society

With information comes responsibility: a responsibility to use information according to appropriate restrictions. Governments, for instance, need to obey legal policies on communicating collected information about private citizens between different departments. The Department of Health might be permitted to share patient data with the Department of Immigration only if a specific warrant has been issued. In recent years, the complexity of policies on information usage has also increased for corporations. Forced by regulations (e.g., GDPR[1]) and public sentiment, technology companies are increasing the transparency of how personal data is used, allowing users to make more fine-grained decisions on how and where their information should flow.

Current systems often do not obey agreed upon *information security policies*, or simply *security policies*, that specify allowed usage of information. To ensure that a system satisfies the desired security policy, one first needs to interpret the security policy, which is expressed in a

[1]Regulation (EU) 2016/679 of the European Parliament on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation).

high-level policy language, in terms of the system behavior. The result of this interpretation is a specific *property* of the system behavior. If the system satisfies this property, then it is expected that the system satisfies the initial policy, too.

Complex security policies on data usage can be interpreted as *information flow properties*. An information flow property is a mathematical specification of how information is allowed to flow between entities making up a system, such as programs, users, inputs, outputs, and storage locations. Consider, for instance, a social-network application where the advertisements shown to users might depend on the social interaction (e.g., joining a group, liking or sharing posts, pages, ads) of their "friends". User Alice might want to specify the security policy that her coworkers (a subset of her friends) should not learn the groups she is a member of. Specifically, the choice of ads shown to her coworkers should not depend on which groups she is a member of. For example, when Alice joins the group "Broccoli Fans," her boss should not start seeing ads about broccoli; otherwise, her boss might infer that there is a broccoli aficionado on staff. So, Alice's initial high-level policy can be interpreted as a specific information flow property: changes in Alice's group membership should not cause changes of ads shown to her coworkers. We refer later to this example property as the Alice-property.

This monograph attempts to match the demand of the digital society for expressing complex data-usage restrictions with the supply of information flow properties proposed in the literature. In doing so, we survey the wide variety of information flow properties that have been formulated within the last four decades, we compare their expressive power, and suggest research directions for a faster convergence between future technological demand and literature supply. Such a large-scale systematization of information flow properties has not been performed before.

1.2 Relation to Privacy, Access Control, and Cryptography

Privacy policies are primarily concerned with restricting the inference of information about individuals. Some privacy policies can be interpreted as information flow properties, which are concerned more broadly

with restrictions on how data may be handled. For example, *use-based privacy policies* (Mundie, 2014), which have the potential to formalize complex regulations (e.g., GDPR, HIPAA[2]), can be interpreted as information flow properties (Birrell and Schneider, 2017). *Differential privacy* (Dwork, 2006), which limits the influence of individual data-samples to the output of an aggregate function, and *contextual integrity* (Nissenbaum, 2010), which restricts information usage based on the context, could be regarded as special cases of use-based privacy (Birrell and Schneider, 2017), and thus be interpreted as information flow properties, too.

Computer systems often employ access control and cryptography to restrict access to sensitive data. However this might not be sufficient to enforce information flow properties. Considering our social-network example, one might attempt to enforce the Alice-property by preventing Alice's coworkers from reading her group memberships. Such prevention can be accomplished by denying read accesses issued by Alice's coworkers (i.e., an access control mechanism), or by encrypting these values with a key unknown to Alice's coworkers (i.e., a cryptographic mechanism). However, preventing Alice's coworkers from reading her group memberships is not enough to enforce the Alice-property. Alice's coworkers should be additionally prevented from reading any value derived from her group memberships, otherwise they may learn something about these memberships. Neither access control nor cryptography can directly restrict access to all these derived values. In fact, one cannot even start addressing this enforcement problem if the information flow property is not made explicit. For this reason, this monograph emphasizes the formal specification of information flow properties, which can concretize the elusive notions of "allowed flow" and "forbidden flow" in terms of system behavior, and clarify when enforcement mechanisms—such as access control or cryptographic mechanisms—can successfully achieve these flow restrictions.

[2]Health Insurance Portability and Accountability Act.

1.3 Information Flow Properties are Hyperproperties

A property of system behavior is commonly a *trace property*: a predicate on a single system execution. A system is said to satisfy a trace property if every possible execution of the system satisfies that trace property. So, in principle, it suffices to examine system executions one-by-one to deduce if there is a "bad" execution that violates the property. For example, an *access control policy*, which stipulates allowed accesses on entities, is interpreted as a trace property, because one "bad" execution where a forbidden access is performed is enough to show that the system does not satisfy this property (an thus the access control policy).

An information flow property is not a trace property, because a single execution is not enough to exhibit a violation. Considering our social-network example, a system execution τ where Alice joins "Broccoli Fans" and her coworkers see broccoli ads does not constitute by itself evidence that Alice-property is violated. If for all other possible executions, Alice's coworkers see those broccoli ads, independently of Alice's group membership, then Alice-property is actually satisfied. But if, in a hypothetical execution τ', Alice does not join "Broccoli Fans" and her coworkers do not see broccoli ads, then Alice-property is indeed violated. The set $\{\tau, \tau'\}$ of executions constitutes evidence that coworkers' ads depend on Alice's group memberships: information flowed from Alice's group memberships to coworkers' ads. Consequently, sets of executions (e.g., $\{\tau, \tau'\}$)—not a single execution—can constitute evidence for violating information flow properties. For this reason, an information flow property is a *hyperproperty* (Clarkson and Schneider, 2010a): a predicate on sets of executions.

1.4 Labels and Security Conditions

An information flow property can be expressed based on *labels*, which are associated with entities and indicate the intended uses of these entities. For example, an entity could be associated with label *Secret*, to signify that this entity stores secret information, and another entity could be associated with *Public*, to signify that it stores public information. Labels are commonly accompanied by a *flow relation*, which signifies how

information is permitted to flow between entities associated with these labels. For instance, a flow relation $\sqsubseteq$ on labels, with $\mathit{Public} \sqsubseteq \mathit{Public}$, $\mathit{Public} \sqsubseteq \mathit{Secret}$, and $\mathit{Secret} \sqsubseteq \mathit{Secret}$, represents that information is allowed to flow from *Public* entities to *Public* entities, from *Public* entities to *Secret* entities, and from *Secret* entities to *Secret* entities. However information is not allowed to flow from *Secret* entities to *Public* entities. Such a flow relation on labels can be considered as a security policy that intuitively describes how flows of information should be restricted. However, this policy is still not precise enough to be rigorously enforced on a system. What is missing is an interpretation of these flow restrictions in terms of the system behavior, in the form of a predicate regarding system executions—an information flow property.

Considering, for instance, a system where inputs and outputs are labeled with *Secret* and *Public*, the information flow restrictions imposed by the above flow relation can be precisely expressed by the following information flow property: Whenever two executions of the system agree on the *Public* inputs (and possibly differ on *Secret* inputs), they should also agree on the *Public* outputs. As desired, this information flow property—a specific predicate on system executions—forbids *Secret* inputs from flowing to *Public* outputs, while it allows all other flows (from *Public* inputs to *Public* outputs, from *Public* inputs to *Secret* outputs, and from *Secret* outputs to *Secret* outputs).

This information flow property is an instantiation of *noninterference* (Goguen and Meseguer, 1982). Noninterference stipulates that information should not flow between entities that are associated with unrelated labels. Noninterference is a *security condition*, since it can be parameterized with different systems, labels, and flow relations. When noninterference is instantiated with a particular system, set of labels, and flow relation, then the result is an information flow property for that system, called an *instantiation of noninterference*. For brevity, one might simply say that a system satisfies noninterference, instead of an instantiation of noninterference.

1.5 Enforcing Information Flow Properties

Information flow control (IFC) mechanisms are used to ensure that a system satisfies an information flow property, which is usually based on a set of labels and a flow relation. Assuming entities are associated with specific labels, a conventional IFC mechanism enforces such a property by propagating labels from one entity to another, along the flow of information. If this label propagation meets an inconsistency (e.g., *Secret* is about to be propagated to an entity associated with *Public*), then the mechanism reports an error. In the general case, enforcing information flow properties is an undecidable problem (Sabelfeld and Myers, 2003b), and thus, an IFC mechanism might conservatively report an error for a system that actually satisfies the desired property.

A wide variety of IFC mechanisms has been presented in the literature. IFC has been extensively studied in the context of programming languages, because restrictions on information usage are ultimately mapped to restrictions on how information flows throughout program executions. In particular, IFC has been applied to functional (e.g., Heintze and Riecke, 1998) and imperative (e.g., Volpano *et al.*, 1996) programming languages, including assembly languages (e.g, Costanzo *et al.*, 2016). IFC has also been used in object-oriented (e.g., Myers and Liskov, 1997), declarative (e.g., Schultz and Liskov, 2013), and concurrent (e.g., Smith and Volpano, 1998) programming languages. For strongly typed programming languages, IFC is usually implemented as part of the compiler, and thus it is statically invoked. For weakly typed programming languages, such as JavaScript, IFC is dynamic (e.g., Austin and Flanagan, 2009) or hybrid (e.g., Moore and Chong, 2011). Model checking methods for IFC have been developed, too (e.g., Clarkson *et al.*, 2014). Sabelfeld and Myers (2003b) discuss information flow properties and enforcement mechanisms in the context of programming languages.

Because programming languages can model a variety of systems, intuition for enforcing information flow policies have been transferred from programs to computer systems more broadly. Hence, IFC has been studied at the hardware level (e.g., Amorim *et al.*, 2014), within operating systems (e.g., Zeldovich *et al.*, 2006) and web browsers (e.g.,

Chong *et al.*, 2007). Also, techniques from IFC are used in the context of distributed systems (e.g., Zeldovich *et al.*, 2008; Liu *et al.*, 2009), blockchains (e.g., Cecchetti *et al.*, 2021), and cyber-physical systems (e.g., Akella *et al.*, 2010).

This monograph does not focus on IFC mechanisms; instead we mainly discuss information flow properties. Focusing mainly on information flow properties is sensible, because an information flow property is usually expressed independently of the enforcement mechanism. This means that the same information flow property can be enforced in several different ways. The rich literature on IFC mechanisms warrants its own survey.

1.6 Scope of the Monograph and Terminology

In general, the formulation of an information flow property for a system involves the selection of the following:

- The entities under consideration, and
- The conditions under which flows between these entities are allowed or forbidden.

The entities are chosen based on the *computational model* and *threat model* for that system: The computational model indicates the entities that are manipulated during system executions; the threat model indicates the entities with which the adversaries interact. So, specifying allowed or forbidden flows between entities amounts to stipulating allowed or forbidden flows between the system and the adversaries. We explore the space of information flow properties by varying the computational model, the threat model, and the expressiveness of the conditions employed to specify restrictions on information flows between entities.

We summarize here the terminology that this monograph employs to systematically discuss the covered literature:

- A *security policy* is a high-level description of desirable system behavior. It is usually specified using a policy language.

- A *label* is a syntactic object that is *associated* with an entity in a system and denotes intended uses of that entity.
- A *flow relation* between labels represents allowed flows between entities associated with these labels. Flow relations usually constitute the language for specifying *information flow policies*, a subset of security policies.
- A *security condition* is a statement parameterized with the labels, the flow relation, and the behavior of the system to specify allowed or forbidden flows between system entities associated with certain labels.
- An *information flow property* is a hyperproperty of the system. It can be the result of instantiating a security condition with certain labels, flow relation, and system.
- An *information flow control mechanism* is an enforcement mechanism that ensures the behavior of a system satisfies an information flow property.

Although we aspire for the above terminology to become lingua franca for the community, researchers have used these terms differently in the past. Some authors (e.g., Denning, 1976) use *security level* or *security class*, instead of label, to refer to syntactic objects that denote intended use for the associated entities. Other authors use term *policy* for the decision to associate certain labels with entities in the system (e.g., Li and Zdancewic, 2005a), or the flow relation between labels (e.g., Sabelfeld and Sands, 2005), or even the way that the flow relation is allowed to change during execution (e.g., Broberg *et al.*, 2015).

For the information flow properties discussed in this monograph, we do not always present exactly their original definition, but rather adapt them to a common formalism. We strive to capture the key ideas and differences, but some subtleties of the definitions may differ due to the change in formalism.

2

Noninterference

To specify allowed (or forbidden) flows of information between entities of a system, we must first understand what information flow is. According to Cohen (Cohen, 1976; Cohen, 1977), information *flows*[1] from entity α to entity β when β depends on or is influenced by α. For example, the salary of an employee flows to the amount of income tax that the employee pays. This is because the income tax depends on the salary: by varying the salary, the income tax varies, too. Inspired by Ashby (1956) and Shannon (1948), Cohen attempts to formalize this intuition. He postulates that information flows from one entity to another if the *variety* of the former is conveyed to the latter. More formally, Cohen proposes *strong dependency* as a definition for information flow:

Definition 2.1 (Strong Dependency)**.** Consider a (deterministic) system H whose inputs include entity α and whose outputs include entity β. Output β *strongly depends on* input α if there exist two executions of H where the inputs differ only for entity α and the output β differs.

In other words, information flows from input α to output β, if varying only input α, while keeping the rest of the inputs fixed, causes output β to change.

[1] Cohen uses the term *information transmission* instead of *information flow*.

Cohen proposes information flow properties[2] that forbid certain flows (i.e., strong dependencies) between entities. Such a property stipulates that variety should not be conveyed between certain entities: changing the value of α should not cause β to change.

Goguen and Meseguer (1982) subsequently used the term *noninterference* to name the requirement that variety should not be conveyed between certain entities. Noninterference is now a widespread security condition that has been employed to express restrictions on information flow for a wide range of computational and threat models.

Here, we introduce the formalism needed to illustrate noninterference for programs. Noninterference imposes restrictions on the behavior of programs, specifying allowed and forbidden flows of information. The behavior of a program C is usually modeled by a set $\mathcal{T}_C$ of *execution traces*, which are sequences of *states*. Finite execution traces in $\mathcal{T}_C$—denoted $\mathcal{T}_C^{\text{fin}}$— represent terminating executions, while infinite execution traces represent diverging executions. A state usually includes a *memory* M that maps variables that appear in program C, denoted as $var(C)$, to values. This memory is updated during execution. Thus, an execution trace $\tau \in \mathcal{T}_C$ is an abstraction of an actual execution of program C. The computational model dictates what information from the actual program execution is represented in the corresponding execution trace. For the *batch computational model* (O'Neill *et al.*, 2006), an execution trace τ contains only the initial memory $\tau.M_0$ and the final memory $\tau.M_\downarrow$ of the modeled execution. If inputs are modeled by variables in the initial memory of the program execution and outputs are modeled by variables in the final memory of the program execution, then the batch computational model essentially focuses only on the inputs and outputs of each program execution—no information about the intermediate computation is captured.

To formulate restrictions on how information flows during program executions, one also needs to define a threat model. Take, for example, a simple threat model where principals provide inputs and observe outputs. Suppose that principals are partitioned into two sets labeled *Alice* and *Coworkers*: set *Alice* contains only Alice and set *Coworkers*

[2]Cohen uses the term *security problem* instead of *information flow property*.

contains Alice's coworkers. A mapping Γ then associates the label *Alice* both with inputs provided by Alice and outputs observed by Alice, and similarly for label *Coworkers*.

Considering the above concrete computational and threat models, we can now precisely express the following restriction: inputs provided by Alice are not allowed to flow to outputs observable by Alice's coworkers. To do so, we employ a security condition known as *relational noninterference* (Clarkson and Schneider, 2010a): for any two finite execution traces $\tau_1, \tau_2 \in \mathcal{T}_C^{\text{fin}}$ of program C whose initial memories $\tau_1.M_0$ and $\tau_2.M_0$ agree on values in variables associated with label ℓ (and possibly disagree on values in variables associated with label ℓ'), their final memories $\tau_1.M_\downarrow$ and $\tau_2.M_\downarrow$ should agree on values in variables associated with ℓ.

Definition 2.2 (Relational Noninterference - RNI(C,ℓ,ℓ',Γ))**.** Given a deterministic program C and a mapping Γ from variables in $var(C)$ to labels in $\{\ell, \ell'\}$, we say that program C *satisfies relational noninterference from* ℓ' *to* ℓ if the following holds:

$$\forall \tau_1, \tau_2 \in \mathcal{T}_C^{\text{fin}}\colon\ \tau_1.M_0 =_\ell \tau_2.M_0 \implies \tau_1.M_\downarrow =_\ell \tau_2.M_\downarrow$$

where

$$M =_\ell M' \triangleq \forall x \in var(C)\colon\ \Gamma(x) = \ell \implies M(x) = M'(x)$$

The result of instantiating RNI with a particular program C, labels *Coworkers*, *Alice*, and mapping Γ, is an information flow property RNI(C, *Coworkers*, *Alice*, Γ), which stipulates that changing the inputs associated with *Alice*, while keeping inputs associated with *Coworkers* the same, should not cause the outputs associated with *Coworkers* to change. Equivalently, executing program C on inputs that are indistinguishable for Alice's coworkers should emit outputs that are indistinguishable for Alice's coworkers, too. So, as desired, RNI(C, *Coworkers*, *Alice*, Γ) implies that inputs provides by Alice should not flow to outputs observed by Alice's coworkers.

The behavior of a program C can now be checked against RNI(C, *Coworkers*, *Alice*, Γ) to understand whether C satisfies that information flow property. Consider, for example, the following program C:

```
if Joined_Broccoli_Group(h) then
    Show_Broccoli_Ads(w)
else
    Show_Carrots_Ads(w)
```

where variable h is the history of Alice's interaction with the social network and w is the news feed observed by Alice's coworker. So, h is associated with label *Alice* (i.e., $\Gamma(h) = Alice$) and w is associated with label *Coworkers* (i.e., $\Gamma(w) = Coworkers$). Notice that when C executes, the ads that Alice's coworkers observe depend on whether Alice has joined a broccoli group. In an execution where Alice has not joined the broccoli group, and thus *Joined_Broccoli_Group*(h) returns *false*, Alice's coworkers are shown carrots ads; in an execution where Alice has joined the broccoli group, and thus *Joined_Broccoli_Group*(h) returns *true*, Alice's coworkers are shown broccoli ads. So, these two possible executions constitute evidence that program C violates information flow property RNI(C, *Coworkers*, *Alice*, Γ).

There might be cases where information should not flow neither from label ℓ to label ℓ', nor from ℓ' to ℓ. So, both RNI(C, ℓ, ℓ', Γ) and RNI(C, ℓ', ℓ, Γ) should hold. McLean (1994) calls this bidirectional flow restriction *separability*.

Notice that RNI(C, ℓ, ℓ', Γ) is not a trace property because examining each execution trace in isolation is not enough to deduce whether RNI(C, ℓ, ℓ', Γ) is satisfied or not. Instead, RNI(C, ℓ, ℓ', Γ) is a hyperproperty, since RNI(C, ℓ, ℓ', Γ) is a predicate on the set $\mathcal{T}_C^{\text{fin}}$ of finite execution traces: any two executions traces τ_1 and τ_2 in $\mathcal{T}_C^{\text{fin}}$ that satisfy $\tau_1.M_0 =_\ell \tau_2.M_0$, should satisfy $\tau_1.M_\downarrow =_\ell \tau_2.M_\downarrow$.

In fact, RNI(C, ℓ, ℓ', Γ) belongs to a special subset of hyperproperties known as *safety hyperproperties* (Clarkson and Schneider, 2010a). A safety hyperproperty Φ is closed under *refinement*, which means that it is closed under the subset relation on traces: if set $\mathcal{T}$ of traces satisfies Φ and $\mathcal{T}' \subseteq \mathcal{T}$ holds, then subset $\mathcal{T}'$ satisfies Φ, too. Consequently, if a

system satisfies Φ, and the behavior of that system is restricted to fewer possible executions, then the resulting system will also satisfy Φ. Being a safety hyperproperty, RNI(C, ℓ, ℓ', Γ) is closed under refinement. However not all information flow properties are closed under refinement, and thus, not all information flow properties are safety hyperproperties. Section 5 includes examples of such properties.

Most information flow properties are hyperproperties, not trace properties. Intuitively, this is because information flow properties can often be expressed as *counterfactual* propositions, which compare (at least) two possible executions. The connection between information flow and counterfactual reasoning is not surprising, because information flow is related to causality and causality is often established by counterfactual propositions (Lewis, 1973): "an event E causally depends on C if, and only if, (i) if C had occurred, then E would have occurred, and (ii) if C had not occurred, then E would not have occurred". Notice that Cohen's strong dependency is indeed a counterfactual proposition: if the value of α were different, then the value of β would have been different, too. An instantiation of noninterference (e.g., RNI)—being the absence of strong dependency—is thus a counterfactual proposition, and consequently, a hyperproperty.

The following sections cover instantiations of noninterference that accommodate richer labels (Section 3), different threat models (Section 4), different computational models (Section 5), and finer-grained notions of allowed flow (Section 6).

The version of record is available at: http://dx.doi.org/10.1561/3300000008

3

Labels

Labels are syntactic objects associated with entities of a system. An information flow policy can describe allowed (or forbidden) flows between entities based on the labels that these entities are associated with. Thus entities associated with the same label are subject to the same flow restrictions. As Montagu *et al.* (2013) put it:

> These labels can be thought of as low-level "micro-policies" for information flow. They do not directly describe the end-to-end security policies that the system's users might care about ("my banking information will never be sent to evil.com"); rather, they capture information flow invariants on specific sensitive values ("this integer and values derived from it should only be visible to the Bank principal").

For this section, labels are assumed to be taken from a given set Λ; no additional assumptions are made about the syntax used to construct labels in Λ. Section 6 will discuss different syntaxes for constructing richer labels that can support more expressive information flow policies.

3.1 Representing Restrictions

Labels represent restrictions on how associated entities can be used. For confidentiality, labels might represent restrictions on who can read some data. For instance, labels *Secret* and *Public* capture such restrictions: *Public* entities (i.e., entities associated with *Public*) store low confidentiality data, and thus, they are allowed to be read by all principals, while *Secret* entities store high confidentiality data, and thus, they are allowed to be read only by a specific subset of principals. Using these labels, an information flow policy can specify that information from *Secret* inputs is not allowed to flow to *Public* outputs.

Other security restrictions can be represented by labels, too. For instance, labels *Untrusted* and *Trusted* can capture integrity restrictions. If an entity is associated with label *Trusted*, then all principals trust that entity; otherwise, no principal trusts that entity. Based on these labels, an information flow policy can specify that information from *Untrusted* inputs, which store low integrity data, is not allowed to flow to *Trusted* outputs, which store high integrity data.

Traditionally, flow restrictions for integrity have been considered the dual to those for confidentiality (Biba, 1977). This is because, for confidentiality, information is not allowed to flow from high to low confidentiality data, whereas for integrity, information is not allowed to flow from low to high integrity data.

Labels can also represent availability requirements. For example, Li *et al.* (2003) consider high availability and low availability data, where high availability data is accompanied by a stronger availability guarantee than low availability data. For instance, high availability data might be stored in multiple replicas and thus, it is highly likely that this data will be available upon request. Whereas, low availability data might be stored in only one server, where one critical server fault might render that data inaccessible. Critical and time-sensitive services had better depend only on high availability data—not on low availability data. This implies that the computation of high availability data should not depend on low availability data, which is an information flow policy studied by Li *et al.* (2003).

Zheng and Myers (2005) later combine integrity and availability requirements into one policy: "with all high availability inputs available, equivalent high integrity inputs will eventually result in equally available high availability outputs". Zheng and Myers (2014) then extend this framework to handle distributed settings, expressing confidentiality, integrity, and availability guarantees for quorum replication schemes.

3.2 Axioms for Flow Relations

As discussed in the Introduction, an information flow policy can be represented as a flow relation $\sqsubseteq$ on a set Λ of labels, such that if $\ell \sqsubseteq \ell'$ holds for labels ℓ and ℓ' in Λ, then information is allowed to flow from ℓ (i.e., entities associated with ℓ) to ℓ'. Notation $\langle \Lambda, \sqsubseteq \rangle$ has been employed to denote the combination of a set Λ of labels and a flow relation $\sqsubseteq$ on these labels.

A flow relation usually satisfies certain axioms. If information is always allowed to flow between entities that are associated with the same label, then flow relation $\sqsubseteq$ is reflexive:

$$\forall \ell \in \Lambda \colon \ell \sqsubseteq \ell.$$

If, in addition, the flow relation is transitive:

$$\forall \ell_1, \ell_2, \ell_3 \in \Lambda \colon \ell_1 \sqsubseteq \ell_2 \ \wedge \ \ell_2 \sqsubseteq \ell_3 \implies \ell_1 \sqsubseteq \ell_3$$

then the flow relation is a *preorder* $\langle \Lambda, \sqsubseteq \rangle$.

One might also want economy of labels: if information is allowed to flow from label ℓ_1 to label ℓ_2 and from ℓ_2 to ℓ_1, then it might be sensible to treat ℓ_1 and ℓ_2 as equivalent labels. This is the antisymmetry axiom:

$$\forall \ell_1, \ell_2 \in \Lambda \colon \ell_1 \sqsubseteq \ell_2 \wedge \ell_2 \sqsubseteq \ell_1 \implies \ell_1 = \ell_2.$$

When adding the antisymmetry axiom to a preorder, the resulting flow relation becomes a *partial order*.

Notice that if a flow relation is a partial order, then it does not contain cycles. This means that for any labels ℓ and ℓ' such that $\ell \sqsubseteq \ell'$ and $\ell \neq \ell'$, then it cannot be the case that $\ell' \sqsubseteq \ell$ holds, too. So it is sensible to say that ℓ' is higher than ℓ. The higher a label is in a flow relation, the more flow restrictions are imposed on the associated

entities. That is because the set of labels that ℓ' can flow to is a subset of the labels that ℓ can flow to, when $\ell \sqsubseteq \ell'$ holds. In general, we say that ℓ' is *at least as restrictive as* ℓ. For confidentiality, entities labeled ℓ are less confidential than entities labeled ℓ'; for integrity, entities labeled ℓ are higher integrity (i.e., more trusted) than entities labeled ℓ'.

In order to enforce an information flow policy, an information flow control mechanism might require the corresponding flow relation to satisfy additional requirements. For example, when combining information from two variables to compute a new value, the enforcement mechanism might need to deduce the label to associate with the new value, from the labels ℓ_1 and ℓ_2 associated with these two variables. Any upper bound of ℓ_1 and ℓ_2 would suffice, as it would obeys flow restrictions both from ℓ_1 and ℓ_2. But a lower upper bound is better, as it will permit as many flows as possible. Moreover, for predictability, we may desire a unique least upper bound. The least upper bound, or *join* of ℓ_1 and ℓ_2, is denoted $\ell_1 \sqcup \ell_2$. Specifically, join $\ell_1 \sqcup \ell_2$ satisfies the following conditions:

- $\ell_1 \sqcup \ell_2$ is at least as restrictive as both ℓ_1 and ℓ_2:

$$\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2 \quad \text{and} \quad \ell_2 \sqsubseteq \ell_1 \sqcup \ell_2.$$

- There is no other label that satisfies the above condition and is less restrictive than $\ell_1 \sqcup \ell_2$:

$$\forall \ell \in \Lambda\colon\ \ell_1 \sqsubseteq \ell \ \wedge\ \ell_2 \sqsubseteq \ell \implies \ell_1 \sqcup \ell_2 \sqsubseteq \ell.$$

A partial order $\langle \Lambda, \sqsubseteq \rangle$ that has a join label $\ell_1 \sqcup \ell_2$ (i.e., least upper bound) for any two labels $\ell_1, \ell_2 \in \Lambda$ is a *join semilattice*.

An information flow control mechanism might also require the existence of a least restrictive label $\bot$, sometimes called the bottom element:

$$\forall \ell \in \Lambda\colon\ \bot \sqsubseteq \ell.$$

Constants in a program are often associated with $\bot$. For confidentiality, this is because the program itself is typically regarded as public information; for integrity, this is because the program is typically trusted.

A finite join semilattice with a bottom element forms a *lattice* of labels (Denning, 1976).[1]

Flow relations that satisfy transitivity might not be always desirable. For example, raw data may flow to a data curator component and the curated data may then flow to some data analysis procedure, but raw data may not directly flow to the data analysis procedure without being curated first. This is a non-transitive flow relation between raw data, data curator, and data analysis. Foley (1989) proposes *reflexive flow policies*, which are flow relations that satisfy reflexivity but not necessarily transitivity. The author also shows how to construct a lattice from a reflexive relation, such that the original flow relation is preserved. Sections 6 and 7 discuss how non-transitive flow relations can be interpreted as information flow properties.

Reflexivity exceptions on flow relations have been studied by Bryce (1997), who points out that "one should distinguish between flows from an entity to itself, and flows among entities of the same class", meaning entities associated with the same label. So, there might be cases where information may flow from an entity x to itself, but it may not flow to other entities that are associated with the same label as x. Bryce shows how finer-grained labels can restore reflexivity and at the same time preserve the restrictions of the original non-reflexive flow relation.

Foley (1991) studies flow relations with *aggregation* and *separation exceptions*. For a flow relation that forms a lattice, if information may flow from ℓ to ℓ'' and from ℓ' to ℓ'', then information may flow from the join $\ell \sqcup \ell'$ to ℓ'' (by the definition of join operator $\sqcup$). A flow relation with aggregation exceptions could specify that information from the aggregate $\ell \sqcup \ell'$ is not allowed to flow to ℓ''. For example, such a flow relation could stipulate that information may flow either from client A or from client B to a particular employee, but information should not flow from both clients to the employee; this is an instance of a *Chinese Wall policy*. A separation exception is the dual of an aggregation exception: $\ell \sqcup \ell'$ is allowed to flow to ℓ'', but neither ℓ nor ℓ' may independently

[1] A lattice has both join and meet operations, where the meet operation provides greatest lower bounds. Some information flow control mechanisms can use the meet operation to find appropriate labels, although the use of the join operation is most common.

flow to ℓ''. One might argue that an encrypted message satisfies this policy: the encrypted message may flow to the public, but neither the original message nor the encryption key are revealed to the public.

Figure 3.1 summarizes axioms discussed in this section that one can select to define a flow relation between labels.

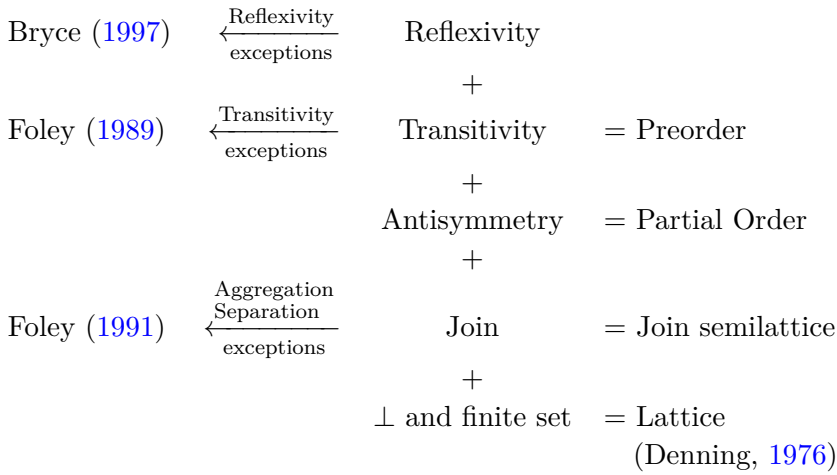

Figure 3.1: Axioms, exceptions, and representative citations for flow relations.

3.3 From Labels to Noninterference

This section describes how a flow relation can be interpreted as an information flow property. In particular, we consider a flow relation that is a partial order and give an instantiation of noninterference that accurately captures the imposed restrictions on information flow. As seen earlier, a flow relation $\sqsubseteq$ defines allowed and forbidden flows between labels of a set Λ. Considering a label $\ell \in \Lambda$, information is allowed to flow from another label $\ell' \in \Lambda$ to ℓ only when $\ell' \sqsubseteq \ell$ holds. We call ℓ-*low* those entities that are associated with such a label ℓ'. So, ℓ-low entities may flow to entities associated with ℓ. We call ℓ-*high* those entities that may not flow to ℓ, because they are associated with a label ℓ' that does not satisfy $\ell' \sqsubseteq \ell$.

Using the terminology introduced above, an information flow policy can specify that, for all ℓ, ℓ-high program inputs should not flow to ℓ-low program outputs. This information flow policy is not specific enough yet to be enforced on programs. So, we instantiate this policy as an information flow property of a deterministic program: whenever two executions of the program agree on initial values of ℓ-low variables (i.e., ℓ-low inputs), and possibly disagree on initial values of ℓ-high variables (i.e., ℓ-high inputs), they should agree on final values of ℓ-low variables (i.e., ℓ-low outputs). This information flow property implies that ℓ-low outputs do not depend on ℓ-high inputs.

The information flow property described above can be parameterized with any partial order $\langle \Lambda, \sqsubseteq \rangle$ and any deterministic program C, to form the following security condition.

Definition 3.1 (Order-based Noninterference - ONI(C, $\langle \Lambda, \sqsubseteq \rangle$, Γ))**.** Given a mapping Γ from variables to labels in Λ, a deterministic program C *satisfies order-based noninterference for* $\langle \Lambda, \sqsubseteq \rangle$ if the following holds:

$$\forall \ell \in \Lambda : \forall \tau_1, \tau_2 \in \mathcal{T}_C^{\text{fin}} :\ \tau_1.M_0 =_\ell \tau_2.M_0 \implies \tau_1.M_\downarrow =_\ell \tau_2.M_\downarrow$$

where

$$M =_\ell M' \triangleq \forall x \in var(C) : \Gamma(x) \sqsubseteq \ell \implies M(x) = M'(x)$$

Notice that ONI can express an integrity or confidentiality information flow property depending on the interpretation of the labels that are considered. Notice also that ONI can express RNI (definition 2.2), since RNI is equivalent to instantiating ONI with a partial order $\langle \{\ell, \ell'\}, \sqsubseteq \rangle$ that satisfies $\ell \sqsubseteq \ell'$.

3.4 Associating Data with Labels

In practice, it is not trivial to select appropriate labels for data (Bergström and Åhlfeldt, 2014). Given that data are considered assets, one needs first to assess the value of these assets. This assessment will then impact the level of protection that should be applied. And these levels of protection will be ultimately represented as labels. So, it seems that associating labels with data needs human intervention. A challenging

open problem would be to find the minimum human intervention required such that entities are correctly labeled (neither overclassified nor underclassified) in an automatic way.

Labels can be associated with new data that enter a system based on certain constraints (Akl and Denning, 1987), such as predicates on data or predicates on labels of other data. For example, information about a new flight reservation that enters the system might need to be labeled *Secret* if the destination of that flight is anywhere in Antarctica. In general, the constraints under consideration should be:

- Consistent: for any pair of constraints, when they are simultaneously satisfied, they do not prescribe conflicting labels for new data.
- Complete: each new data piece should be mapped to a label.

Ideally, constraints should be solved such that the satisfying labels impose minimum restrictions on the associated data (Dawson *et al.*, 1999). These constraints can also be used to associate desired labels to a collection of data, and thus this approach can address the *aggregation problem* (Meadows, 1990), which is when "two or more data items are considered more sensitive together than they are separately". Notice that the constraints discussed above are solved once for each new data item. In general, such constraints might depend on the execution state of a system and be solved dynamically for all data items (existing and new ones). In this way, the label of a given data item might change throughout system execution. Section 6 discusses frameworks that could support this general case.

A common design decision that one has to make is the level of granularity at which information is labeled. In general, finer-grained labeling schemes offer greater control over information, but might require more elaborate enforcement mechanisms. Taking a database as an example, will information be labeled at the granularity of entire tables, rows within a table, or individual values within a row? Clearly, labeling rows within a table is finer grained than labeling only tables, and labeling values within a row is still finer. Going even lower in the implementation stack, one can associate memory pages, cache lines, and registers with labels (e.g., Ferraiuolo *et al.*, 2018).

Also, will labels be associated with the containers of data or the data itself (Woodward, 1987; Hartman, 1988)? Associating a label with the container implies that any data stored in that container is considered to be associated with that same label. Whereas, associating a label with the content implies that each piece of information stored in a container might be associated with a possibly different label. The question of whether labels are associated with the container or the contents of the container may be considered either an enforcement issue or a policy issue depending on the setting. So, such a labeling decision might be related to expressing the desired restrictions on the use of information, or it might be related to how to enforce the desired restrictions.

4

Threat Model

An information flow property for a system is typically based—explicitly or implicitly—on a *threat model*, which formally characterizes how principals interact with the system. The information flow properties presented so far (i.e., instantiations of RNI and ONI) considered a simple threat model, where principals provide initial values of program variables and observe final values of program variables. The stronger the threat model is, the richer the set of interactions become, and thus, the more opportunities there are for information to flow between principals and the system. In this section we see how noninterference can be extended to proscribe flows under a variety of commonly employed threat models. We focus primarily on information flow properties for several threat models that have been proposed since 2003; Sabelfeld and Myers (2003b) cover relevant earlier literature.

4.1 Information Channels

An *information channel* is an entity that conveys or transmits information about the system behavior and may be observable by principals. Adopting the terminology of Lampson (1973), information channels can be split into *legitimate channels* and *covert channels*. Legitimate

channels are those intended to convey information to observers. Examples of legitimate channels include network ports, files, and printed program results. Covert channels are illegitimate channels, which are not intended to convey information. Execution time of a program and heat emission of a system are such examples.

Covert channels are further categorized based on how adversaries exploit them. A covert channel exploited by a *passive* adversary, who simply observes the conveyed information, is called a *side channel*. Side channels are distinguished from other covert channels, where an *active* adversary exploits them to signal information from the system to external parties.

In what follows, we explore threat models that consider different channels and describe information flow properties that restrict flows through those channels.

4.2 Termination

The termination behavior of a program can form a covert channel, known as *termination channel*. To illustrate, consider the following program C, which terminates depending on the value stored in sensitive variable h:

$$\begin{array}{l} \textbf{if } h > 0 \textbf{ then} \\ \quad \textbf{while } \mathit{true} \textbf{ do skip} \\ \textbf{else} \\ \quad l := 2 \end{array} \tag{4.1}$$

If condition $h > 0$ holds, then the program diverges, otherwise the program terminates. So, the termination behavior of the program depends on the sensitive variable h. Thus, under a threat model where the adversary can observe the termination behavior of that program, information can flow from sensitive data stored in h to that adversary, through the termination channel.

An information flow property can specify whether information is allowed to flow through a termination channel. For example, ONI (Definition 3.1) allows any information to flow through the termination channel, since ONI considers only terminating execution traces and

imposes no restrictions on diverging traces. Taking program C in (4.1) as an example, and assuming that h is associated with label *Secret* and l is associated with label *Public*, we have that information flow property ONI(C, $\langle\{Public, Secret\}, \sqsubseteq\rangle$, Γ) is satisfied: Whenever two finite executions of C agree on initial values of *Secret* variable h, they also agree on final values of *Public* variable l. So, program C satisfies ONI, even though C leaks sensitive information through the termination channel.

ONI can be extended to consider all traces (including diverging traces), and thus, proscribe leaks through the termination channel. This extension of ONI is called *termination-sensitive noninterference* (TSNI) (Volpano and Smith, 1997). According to TSNI, high confidentiality inputs of a program should not flow to low confidentiality outputs and to the termination behavior of the program. Specifically, for any label $\ell \in \Lambda$, if two executions of a deterministic program agree on initial values of ℓ-low variables (see Section 3.3 for this terminology), and possibly differ on initial values of ℓ-high variables, then these two executions should either both diverge, or both terminate agreeing on final values of ℓ-low variables. Consequently, changing values in ℓ-high variables does not cause the termination behavior of the system to change, and thus leaking sensitive information through the termination channel is avoided.

Definition 4.1 (Termination-Sensitive Noninterference - TSNI(C, $\langle\Lambda, \sqsubseteq\rangle$, Γ))**.** Given a mapping Γ from variables to labels in Λ, a deterministic program C *satisfies termination-sensitive noninterference for* $\langle\Lambda, \sqsubseteq\rangle$ if the following holds:

$$\forall \ell \in \Lambda : \forall \tau_1 \in \mathcal{T}_C^{\text{fin}} : \forall \tau_2 \in \mathcal{T}_C : \tau_1.M_0 =_\ell \tau_2.M_0 \implies \tau_2 \in \mathcal{T}_C^{\text{fin}} \ \wedge \ \tau_1.M_\downarrow =_\ell \tau_2.M_\downarrow$$

where

$$M =_\ell M' \triangleq \forall x \in var(C) : \Gamma(x) \sqsubseteq \ell \implies M(x) = M'(x)$$

4.3 Time

Sensitive information might flow to adversaries that observe the timing behavior (i.e., execution time) of a program. Such *timing channels* can lead to serious security vulnerabilities. For example, in some cryptosystem implementations, the time taken to encrypt a message depends on the number of 1's in the encryption key, and thus the time it takes to encrypt a message reveals information about the encryption key (Kocher, 1996). So, sensitive information about the encryption key is leaked to the adversaries through the timing channel.

An information flow property can specify whether information may flow through a timing channel. To do this, consider an auxiliary variable T that models the execution time. An extension of ONI (definition 3.1) can then proscribe flows from sensitive inputs to T: it suffices to extend definition $M =_\ell M'$ with the conjunct $M(T) = M'(T)$. The resulting security condition then stipulates: For any label $\ell \in \Lambda$, if two executions of a deterministic program agree on initial values of ℓ-low variables (see Section 3.3 for this terminology), and possibly differ on initial values of ℓ-high variables, then these two executions should either both diverge, or both terminate agreeing on final values of ℓ-low variables and on the execution time T. This extension of ONI, which is called *time-sensitive noninterference* (Kashyap *et al.*, 2011), proscribes the flow of information from ℓ-high variables to the execution time T, and thus, it avoids the creation of a timing channel to the adversaries.

Similar to timing channels, information flow properties can express restrictions on flows to other covert channels, such as acoustic-emanations channels (e.g., Guri *et al.*, 2017) and heat-emission channels (e.g., Masti *et al.*, 2015). A general approach is to introduce an entity, such as an auxiliary program variable, that models the covert channel under consideration, assume that this variable is observable by the adversary, and require that no sensitive information is allowed to flow to that variable. The challenge is to ensure that the model is accurately capturing the covert channel; especially if this covert channel is based on physical characteristics of the system, such as heat emission.

4.4 Interaction

Applications can emit streams of outputs during execution. Such steams might leak sensitive information to observers. Consider, for example, the following program:

$$\begin{aligned} &\textbf{if } h > 0 \textbf{ then} \\ &\quad l := 2;\ l := 3;\ l := 4 \\ &\textbf{else} \\ &\quad l := 2;\ l := 3 \end{aligned} \tag{4.2}$$

Depending on the sensitive information stored in *Secret* variable h, the observers of *Public* variable l receive different output streams (they observe $2, 3, 4$ when $h > 0$, and $2, 3$ when $h \not> 0$). So, sensitive information about h is leaked to *Public* principals, who observe the *Public* variable l. *Progress-sensitive noninterference* (PSNI) (Askarov *et al.*, 2008; Askarov and Myers, 2010) can proscribe such leaks. PSNI stipulates that executing a program on memories indistinguishable for *Public* principals should produce indistinguishable output streams for these principals, too. PSNI also requires that the termination behavior of these executions should be indistinguishable for *Public* principals. Askarov and Myers (2010) propose *progress-insensitive noninterference* (PINI) to relax that last requirement on the termination behavior. So, under PINI, sensitive information might leak to *Public* principals through the termination behavior of a program.

Principals might provide streams of inputs during program execution, too. Bohannon *et al.* (2009) study *reactive* programs, which produce streams of outputs as a response to streams of inputs provided by principals during execution. A reactive program satisfies *reactive noninterference* when input streams that are indistinguishable for a principal lead to output streams that are indistinguishable for that principal, too.

Suitable information flow properties for reactive programs had been studied early on in the literature. *Noninterference*, proposed by Goguen and Meseguer (1982), stipulates that for any execution trace τ, there will always be another execution trace τ', such that τ' has no high confidentiality inputs and agrees on low confidentiality inputs and outputs with

τ. In this way, an adversary that observes low confidentiality inputs and outputs would not know whether these observations were due to τ, which actually involves sensitive information, or τ', which involves no sensitive information. So, with this security condition, which we will be calling *Goguen–Meseguer Noninterference* (GMNI), no sensitive information is leaked to the adversary. Alternatively, *noninference* (O'Halloran, 1990) stipulates that removing all high confidentiality inputs and outputs from an execution trace should result in a valid execution trace. *Generalized noninference* (McLean, 1994) relaxes noninference by requiring that removing only the high confidentiality inputs (not high confidentiality outputs) from an execution trace results in a valid execution trace. Allen (1991) and Zakinthinos and Lee (1997) provide a comprehensive comparison of these security policies.

Inputs that a principal provides to a program might depend on previous inputs and outputs that this principal has witnessed. *Strategies* (Wittbold and Johnson, 1990) can be employed to model this behavior. Strategies are mappings from sequences of inputs and outputs to the next input that the corresponding principal will provide to the program. O'Neill *et al.* (2006) build on *nondeducibility on strategies*, a security condition proposed by Wittbold and Johnson (1990), to propose an instantiation of noninterference that proscribes leaks to adversaries modeled by strategies.

The more powerful the adversaries, the more sophisticated the strategies that they employ. Thus, a system that satisfies noninterference against a certain set of strategies, might leak information against a different one. Clark and Hunt (2008) study the relation between instantiations of noninterference that are expressed against different sets of strategies. The authors proved that noninterference against the set of all strategies AS (including non-deterministic ones) is equivalent to noninterference against the set of all deterministic strategies DS. So, a system that satisfies noninterference against DS, it also satisfies noninterference against AS. The authors also consider set SS of all *stream strategies*: strategies that return the next element in a predetermined stream without depending on previously observed inputs or outputs. For deterministic programs, they show that noninterference against SS is equivalent to noninterference against AS. Consequently,

for deterministic programs, it suffices to model adversaries' behavior with a stream.

Programs can be used to model computable strategies. In this case, the execution of the original system and the execution of the program that represents the attacker's strategy might be regarded as concurrent processes. Focardi and Rossi (2002) follow this intuition by modeling a hostile environment with a program that executes in parallel with the program of interest. The authors express a security condition appropriate for dynamic hostile environments, called *Persistent Bisimulation-based Non Deducibility on Compositions* (P_BNDC). This security condition stipulates that every execution step of program C in any dynamic hostile environment is indistinguishable, for the adversary, from executing C alone. This statement quantifies over every hostile environment for every execution step, which implies that the information flow property can handle environments that might change during system execution.

Probabilistic strategies have been employed by Tedesco *et al.* (2016), as a way to model faults that adversaries induce on a system. In particular, the authors consider adversaries that induce faults on a restricted set of memory locations with a probability distribution that depends on these locations and on the observations the attacker has made so far. The authors introduce *probabilistic fault-resilient non-interference* (PFRNI) as a security condition appropriate for this threat model: "a system is secure in the presence of faults if, for any attacker influencing the injection of faults, the probability of a given public output is independent of the secrets held by the system."

4.5 Program Code

The threat models discussed so far assume that (i) every principal knows the entire program code and (ii) no principal can modify it. These assumptions are incorporated in the corresponding security conditions. In particular, the pairs of the executions that are considered by these security conditions (e.g., ONI) are generated by the same program C. This means that all principals know that program C is the one that is being executed. However, there are cases where parts of program code should remain secret to some principals, following an approach

known as "security by obscurity". Proprietary code is such an example. An interesting problem is to propose information flow properties for programs that are kept secret to certain principals.

In addition to knowing the program code, adversaries might also modify the code and data processed by that code. Such *active adversaries* (Zdancewic and Myers, 2001) are considered by Fournet and Rezk (2008). Given a program C, an active adversary a may write some code, which reads and modifies certain variables, and place it in a predetermined hole within C. Program C satisfies *noninterference against active adversary* a (ANI), if for any code that active adversary a decides to insert into program C, sensitive information should not flow to observations made by a.

Zdancewic and Myers (2001) propose *robustness* to stipulate that an active adversary learns nothing more than what a passive adversary would have learned about confidential inputs of a certain program. Essentially, robustness implies that if a program satisfies noninterference against a passive adversary, then it should also satisfy noninterference against an active adversary. Robustness is extensively discussed in Section 6.2.

4.6 Views

A threat model might allow different principals to observe different aspects of system behavior. These aspects can be modeled as *views* of a system, which are projections of relevant information from execution traces to certain principals. For example, a view of a system could be a memory projection that includes only variables associated with certain labels. In general, a view can be defined as an arbitrary function applied to the behavior of a system.

Security conditions can describe allowed or forbidden flows between different views of a system. *Nondeducibility* (Sutherland, 1986) is such a security condition that allows no information to flow between two views: "given two views f and g on a trace set T, if no possible f-observation excludes any possible g-observation, then the views are non-deducible". Comparing to noninterference, McLean (1990a) observes that nondeducibility is symmetric, which means that no information

flows from view f to view g if and only if no information flows from g to f. For example, if f is instantiated as projecting only the values of *Secret* variables in a memory and g as projecting only the values of *Public* variables, then nondeducibility implies that information is not allowed to flow from *Secret* to *Public*, nor from *Public* to *Secret*. This symmetry, which might lead to more restrictive information flow properties than desired, could be avoided using methods proposed by Halpern and O'Neill (2002). Focardi and Gorrieri (1995) discuss and compare multiple instantiations of nondeducibility and noninterference. More recently, nondeducibility has been used as a desirable security condition in the context of cyber-physical systems (Bohrer and Platzer, 2018).

Hughes and Shmatikov (2004) propose a special instance of nondeducibility, called *opaqueness*. As described by Schoepe and Sabelfeld (2015), "a predicate on system behaviors is opaque if for any behavior that satisfies the predicate, there is another behavior that is indistinguishable by the attacker but where the predicate no longer holds". Hughes and Shmatikov express anonymity, unlinkability, and privacy guarantees in terms of opaqueness. So, given the relations that are formed above, anonymity, unlinkability, and privacy could be interpreted as information flow properties.

Figure 4.1 summarizes the threat models discussed in this section, accompanied by representative security conditions that prevent leaks to corresponding adversaries.

The adversary can:	**Example security conditions**
Observe termination	Termination-sensitive noninterference
Observe time	Time-sensitive noninterference
Observe output stream	Progress-sensitive noninterference
and provide input stream	Reactive noninterference, GMNI, noninference, generalized noninference
and use input strategies	Nondeducibility on strategies
and be a concurrently executed program	P_BNDC
Write program code	Noninterference against active adversary
Observe views of system behavior	Nondeducibility, Opaqueness

Figure 4.1: Security conditions for different threat models.

5

Computational Models

Information flow properties describe allowed or forbidden flows between entities of a system. These entities are identified based on a computational model, which abstracts system functionality, and a threat model, which describes how an adversary might interact with the system. A computational model can capture the implementation details of a system more or less faithfully. But the more faithfully the computational model captures these details, the more it may expose opportunities for information to flow between system entities and the adversary. This section considers suitable information flow properties that restrict these flows.

5.1 Nondeterminism

A crucial design decision for defining a computational model is whether this model is deterministic or nondeterministic: whether identical system inputs lead to identical outputs or possibly different outputs. Nondeterminism naturally arises in practice, for example due to concurrency or due to underspecified operations. For instance, considering execution time as a modeled output, it is apparent that in practice, multiple executions of the same program under identical inputs may have different

execution times. And this nondeterminism on the execution time can be considered a consequence of underspecified executed commands.

Sensitive information might leak to adversaries through the nondeterministic behavior of a system. Consider, for example, the following program:

$$\begin{aligned} &\textbf{if } h > 0 \textbf{ then} \\ &\quad l := 2 \;||\; l := 3 \\ &\textbf{else} \\ &\quad l := 2 \;||\; l := 4 \end{aligned} \tag{5.1}$$

where $||$ denotes a nondeterministic choice between two commands, h is a *Secret* variable, and l is a *Public* variable. If at the end of the execution l is 3, then it can be deduced that $h > 0$ holds; if at the end of the execution l is 4, then $h \not> 0$ holds. So, sensitive information is leaked from *Secret* h to *Public* l.

Observational determinism (Roscoe, 1995; Zdancewic and Myers, 2003) can proscribe such leaks. Observational determinism is essentially ONI (definition 3.1) for nondeterministic programs: executing a program on inputs that are indistinguishable for an adversary should yield indistinguishable outputs. So, from the adversary's points of view, a nondeterministic system should behave deterministically. As expected, program (5.1) does not satisfy ONI.

Some may find observational determinism too restrictive, since it forbids any nondeterminism on *Public* observations, even if this nondeterminism conveys no sensitive information. Consider, for instance, a modification of (5.1), where the condition here involves *Public* l' instead of *Secret* h:

$$\begin{aligned} &\textbf{if } l' > 0 \textbf{ then} \\ &\quad l := 2 \;||\; l := 3 \\ &\textbf{else} \\ &\quad l := 2 \;||\; l := 4 \end{aligned} \tag{5.2}$$

Notice that no sensitive information is leaked when executing this program. However (5.2) does not satisfy observational determinism, because executing the program twice with the same value of *Public* l'

(e.g., a value that satisfies $l' > 0$) might yield different values of *Public* l (e.g., l can be 2 or 3).

On the other hand, *possibilistic* information flow properties allow some nondeterminism on *Public* observations, provided this nondeterminism does not leak sensitive information. *Generalized noninterference* (GNI) (McCullough, 1988) is a possibilistic security condition, since it requires the possibility of the occurrence of a *Public* output to be independent of *Secret* inputs. GNI stipulates that executing a program on different *Secret* inputs should not cause the set of possible *Public* outputs to change. As desired, program (5.2) satisfies GNI, since the set of possible values assigned to *Public* l does not depend on any *Secret* inputs.

McLean (1994) interprets GNI as follows: *Secret* inputs do not constrain the possible *Public* observations (e.g., inputs and outputs). Specifically, for a system that satisfies GNI, there exists an execution trace for every possible combination of *Secret* inputs and *Public* observations. Equivalently, for every two execution traces of the system, there exists a third one that combines the *Secret* inputs of the first one and the *Public* observations of the second one. The following definition of GNI is due to Clarkson and Schneider (2010a) and based on McLean's version of GNI.

Definition 5.1 (Generalized Noninterference - GNI(C, $\{\ell, \ell'\}$, Γ))**.** Given a mapping Γ from variables to labels in $\{\ell, \ell'\}$, a nondeterministic program C *satisfies generalized noninterference from* ℓ *to* ℓ' if the following holds:

$$\forall \tau_1, \tau_2 \in \mathcal{T}_C^{\text{fin}} \colon \exists \tau_3 \in \mathcal{T}_C^{\text{fin}} \colon \tau_1.M_0 =_\ell \tau_3.M_0 \ \wedge \ \tau_2.M_0 =_{\ell'} \tau_3.M_0 \\ \wedge \ \tau_2.M_\downarrow =_{\ell'} \tau_3.M_\downarrow$$

In contrast to observational determinism, possibilistic information flow properties do suffer from *refinement attacks*, where an adversary might drive the system to expose only a subset of the possible outputs, causing sensitive information to leak. Considering GNI, it is the existential quantifier in definition 5.1 that prevents closure under refinement, and thus, enables a refinement attack: if a set $\mathcal{T}$ of execution traces satisfies GNI, then a subset of $\mathcal{T}$ might not necessarily contain sufficient witness traces to satisfy the existential quantifier. So, given a system

that satisfies GNI, an attacker might preclude some possible executions of the system and force it to leak sensitive information and violate GNI.

Possibilistic security conditions, such as GNI, do not consider the probability distributions of observed values. However, if sensitive information influences the distribution of observed values, some information might be leaked. Consider, for example, the parallel composition of command C_1 : "**while** $s > 0$ **do** $s := s - 1$ **end**; $p := 2$" and command C_2 : "$p := 1$", where s is a *Secret* variable storing a non-negative integer and p is a *Public* variable. The overall concurrent program satisfies GNI from *Secret* to *Public*, because the possible final values of *Public* variable p are either 1 or 2, independently of the initial *Secret* value of s (for any value of s, C_2 might execute before or after C_1, so the final value of p might be 2 or 1, correspondingly). However, Smith (2006) points out that the probability for p to be 2 at the end of execution does depend on the initial value of s: the larger the initial value stored in s, the more likely is for C_1 to complete execution last, and thus, the more likely is for the final value of p to be 2. So, s leaks to the probability distribution of p. Volpano and Smith (1999) propose *probabilistic noninterference* to proscribe such leaks. This security condition proscribes leaks through timing channels, too. Later, Smith (2006) relaxes probabilistic noninterference to *weak probabilistic noninterference*, which allows leaks through timing channels.

Figure 5.1 summarizes the comparison between observational determinism, possibilistic, and probabilistic security conditions, which are appropriate for nondeterministic systems. This comparison examines (i) whether the security condition allows some nondeterminism to be observable by the public (provided this nondeterminism does not leak sensitive information), (ii) whether it defends against refinement attacks, and (iii) whether it can prevent leakage through the probability distribution of the outputs, which are observed by the adversaries.

5.2 Composition of Systems

A system can be modeled as a composition of subsystems. Such subsystems can be connected with each other using a combination of the composition constructs below:

– Sequential, where the output of one subsystem becomes the input for the next one,

– Parallel, where an input is copied to several subsystems and the outputs of the subsystems are merged into one output, and

– Feedback, where outputs of a subsystem become its own inputs.

	Allows public non-determinism	Defends against refinement attacks	Defends against leaky output distributions
Observational Determinism	X	✓	✓
Possibilistic	✓	X	X
Probabilistic	✓	?	✓

Figure 5.1: Comparing security conditions for nondeterministic computational models. Note that it is unclear what constitutes a refinement attack against a probabilistic system.

McCullough (1988) points out that the mere composition of deterministic components might give rise to nondeterminism. For example, under parallel composition, outputs from system components might be nondeterministically merged into the output of the whole system. For this reason, the author proposes GNI to proscribe leaks through this nondeterministic behavior.

To facilitate the modular development of secure systems, researchers have studied the compositionality of proposed information flow properties. In general, a compositionality problem (McLean, 1994) is formulated as follows: given a system that satisfies property Φ and a system that satisfies property Φ', what is the property that is satisfied when combining these two systems under a certain composition construct? Zakinthinos and Lee (1996) study the compositionality of GNI, showing that the only possible interconnection that can cause a system constructed from GNI secure components not to satisfy GNI occurs under a feedback composition of two components.

McCullough (1988) studies the compositionality of nondeducibility, after showing that GNI implies nondeducibility (but not vice-versa). He concludes that nondeducibility is even less composable than GNI. The author also shows that if nondeducibility is strengthened by forbidding *unsolicited write–ups* (i.e., "a high–level output at a time when there has been no high–level input requesting it"), then compositionality is restored.

Zakinthinos and Lee (1995) studied whether the satisfaction of GMNI by certain system components imply the satisfaction of GMNI by the composition of these components. Assuming that an output from one system component can immediately become an input to another system component (and thus avoiding nondeterminism), the authors prove that GMNI is always preserved by a feedback-free composition of systems. Additionally, the authors show that there are conditions under which GMNI can be preserved by a feedback composition of systems, too.

Mantel (2002) derives compositionality results for several security conditions (e.g, nondeducibility, separability, noninference, generalized noninference, generalized noninterference). Later, Mantel *et al.* (2011) propose *SIFUM-security* (i.e., Secure Information Flow Using Modes) as a security condition that can be preserved under parallel composition. This security condition enjoys increased permissiveness by incorporating assumptions (i.e., assume/guarantee statements) about the protocol that concurrent threads employ to access shared resources.

5.3 Concurrency

In concurrent systems, processes are interleaved, and thus information might flow from one process to another through this interleaving. Sabelfeld and Myers (2003b) give an overview of information flow properties that describe which of these flows are allowed or forbidden. Here, we mainly focus on information flow properties proposed since the publication of that paper.

Interactive Processes

Rafnsson and Sabelfeld (2014) consider concurrent processes that interact with each other through input and output channels. The proposed security conditions are possibilistic versions of PSNI and PINI (discussed in Section 4.4): the possibility of the occurrence of a sequence of observable outputs should be independent of sensitive input sequences. The authors also study conditions under which possibilistic PINI and possibilistic PSNI are preserved under parallel composition of processes.

Scheduling

In some concurrent systems, a scheduler decides which thread will be executed next. But this decision might leak sensitive information. Consider, for example, the following concurrent program, which is discussed by Zdancewic and Myers (2003):

$$l := \mathit{true} \mid l := \mathit{false} \mid l := h$$

where h is a *Secret* variable that stores a Boolean and l is a *Public* variable, whose final value is observed by the adversary. This program satisfies GNI, because the set of possible final *Public* values observed by the adversary (i.e., $\{\mathit{true}, \mathit{false}\}$) is independent of the value in *Secret* h. However, if we consider a scheduler that always executes the third thread last, then the *Secret* value will leak to the adversary. Here, the scheduler refines the behavior of the system by allowing fewer possible executions, and thus, it exposes a leak.

If sensitive information is used to make scheduling decisions, then a scheduler might leak this information, too. Russo and Sabelfeld (2006) give the following example: if the number of some new threads added in the thread pool depends on a secret and the decision to schedule the next thread depends on the number of threads in that pool, then the secret might leak to subsequent public observations.

An information flow property can be used to proscribe leaks, against a specific scheduler, a class of schedulers, or all schedulers. Smith (2006) proposes a probabilistic information flow property that considers a *uniform* scheduler, which picks the next thread to be executed uniformly at random. Sabelfeld and Sands (2000) and Russo and Sabelfeld (2006)

regard a scheduler as a special program that runs concurrently with the rest of the threads. The authors express the proposed information flow properties with respect to a class of schedulers that satisfy noninterference and some additional security constraints. Then, a concurrent program is considered secure if executing it under any of those schedulers does not leak sensitive information. Mantel and Sudbrock (2010) propose information flow properties for *robust schedulers*, a class of schedulers where the scheduling of public threads (i.e., threads that might modify public variables) does not depend on sensitive threads (which do not modify public variables).

Memory Model

Under concurrent execution, threads issue load and store commands to a shared memory. The memory model defines the order in which these commands become visible to each thread. For example, according to the *sequential consistency* (SC) memory model, all commands appear to be executed in a sequential order that obeys the order in which each thread issues its own commands. *Total store order* (TSO) is a weaker memory model, which is exposed by many hardware architectures and programming languages, as it allows more efficient implementations than SC. Vaughan and Millstein (2012) show that the memory model affects whether a certain concurrent program satisfies a given information flow property. In particular, the authors give a concurrent program that satisfies possibilistic noninterference when executed under SC and violates possibilistic noninterference when executed under TSO. They give a concurrent program for the opposite case, too. So, possibilistic noninterference under SC is incomparable to possibilistic noninterference under TSO.

In addition to SC and TSO, Mantel *et al.* (2014) study the IBM 370 and the *partial store order* (PSO) memory model and show that possibilistic noninterference for PSO, SC, and TSO are pairwise incomparable. Note that these results arise because the different memory models give different operational semantics to the same program. That is, the set of possible executions of a given program depends on the memory model used for execution. By contrast, if a programming language provides

concurrency abstractions that ensure well-defined semantics regardless of the memory model of the underlying execution platform, then the incomparability of noninterference under different memory models is not relevant for the specification of security in that programming language.

Speculation

For performance reasons, a processor proactively executes commands under the speculation that this transient execution will be actually invoked in the future. When a processor misspeculates, the architectural state (e.g., registers, memory) is reset to values that existed before the transient execution started, but the microarchitectural state (e.g., cache or branch predictors) is not. So executions that follow a misspeculation can exfiltrate secret information that might have been encoded in the microarchitectural state by that transient execution. Thus, under a misspeculation, the microarchitectural state might function as a covert channel that can leak sensitive information to adversaries. Cheang *et al.* (2019) present *trace property-dependent observational determinism* (TPOD) as an extension of observational determinism (discussed in Section 5.1) that proscribes such leaks. Guarnieri *et al.* (2020) propose *speculative noninterference* to specify that "speculatively executed instructions do not leak more information into the microarchitectural state than what is leaked by the standard, non-speculative semantics".

6

Reclassification

The information flow policies presented so far imply that if a piece of data x is associated with a label ℓ, then any information derived from x is always considered to be associated with ℓ or a more restrictive label than ℓ (according to a flow relation $\sqsubseteq$). This is a restrictive regime. In practice, there are conditions under which derived information is expected to be associated with different labels than those prescribed by the original label of the original data. Such relabeled information is thus allowed to flow to different entities than those prescribed by the original label. This change of allowed flows is called *reclassification* (Denning, 1975).

Reclassifications specified by confidentiality and integrity information flow policies are described using specialized terminology. A *declassification* occurs when confidential information is allowed to flow to lower confidentiality entities. For example, consider a password checker that compares a user input with a secret password and subsequently outputs the result to the public. Here, information about the secret password (whether it equals the user input) is allowed to flow to the public, and thus, this information is allowed to be declassified. An *erasure* (Chong and Myers, 2008) occurs when information is further restricted to flow

only to higher confidentiality entities. For example, after the completion of an electronic payment, the credit card details of the user should no longer flow to the payment system; these details should actually be erased from the system. For integrity, an *endorsement* (Zdancewic *et al.*, 2001) occurs when lower integrity information is allowed to flow to higher integrity entities. For instance, when the untrusted user input is sanitized (e.g., by removing or escaping certain characters), then the sanitized data can be endorsed and stored in the system. With a *deprecation* (Kozyri and Schneider, 2020), information becomes less trusted. For example, data signed by a key, and any information derived from this data, becomes deprecated when that key is compromised. In general, a reclassification, such as declassification and endorsement, that allows an otherwise forbidden flow towards less restrictive labels (with respect to a flow relation) is called a *downgrade*. Whereas, a reclassification, such as erasure and deprecation, that requires a flow towards more restrictive labels is called an *upgrade*. Figure 6.1 summarizes the terminology discussed above.

	Confidentiality	**Integrity**
Downgrade	Declassification	Endorsement
Upgrade	Erasure	Deprecation

Figure 6.1: Terminology for reclassification

Sabelfeld and Sands (2009) survey information flow properties that can describe declassification. The authors propose healthiness criteria for such properties. They also categorize conditions that permit declassifications using four dimensions:

- *What* information may be declassified. For example, one might allow only the result of the majority function to be declassified to the public but not the individual votes.
- *Where* the declassification may occur. An interpretation of this dimension uses program code locality, where a secret may be declassified only at a particular program point. Another interpretation is based on the label lattice locality, where a declassification

stipulates the label with which some declassified data is ultimately associated with.

- *When* the declassification may occur. For example, a secret bid is declassified when the auction closes.
- *Who* may perform the declassification. A secret, for instance, may be declassified only under the authority of a particular principal.

This last dimension is discussed in the next section, which concerns the interplay between information flow policies and authorization.

The current section focuses on the remaining three dimensions (i.e., what, where, when) and discusses research mainly conducted after the publication of the above survey. In particular, the research discussed in this section proposes reclassification conditions and appropriate information flow properties, explores knowledge-based definitions for these properties, and presents additional healthiness criteria for reclassifications.

6.1 Reclassification Conditions

Information flow policies and properties can describe conditions under which a reclassification may or must occur. We examine such reclassification conditions and discuss which of the first three dimensions (i.e., what, where, when) each condition encompasses.

6.1.1 Trusted Processes

There are cases where information is allowed to flow from one entity to another only if this flow is mediated by a *trusted process* (Rushby, 1981). For example, a secret message is allowed to flow to a public channel only if this message is first encrypted. In this example, the encryption module is the trusted process that declassifies information from the secret message to the public. Notice that the flow relation for this information flow policy is intransitive: secret information is allowed to flow to the encryption module, the result of the encryption module is allowed to flow to the public, but secret information is not allowed to flow directly to the public channel (i.e., without the mediation of

the encryption module). Noninterference instantiations discussed in previous sections interpret flow relations that are preorders, which are by definition transitive. Thus those noninterference instantiations are not suitable for intransitive flow relations.

Rushby (1992) proposes *intransitive noninterference* as a security condition that captures intransitive flow relations for a system that enables interaction between multiple users. If information is allowed to flow from user u to user v only when mediated by user w, intransitive noninterference stipulates that v's view of the system should look the same as if one had removed all actions from u that have not had an opportunity to flow to v via w—actions from u without a subsequent action from w. Engelhardt *et al.* (2012) show how intransitive noninterference can be extended to nondeterministic systems (e.g., distributed systems). Later, Lu and Zhang (2020) instantiate intransitive noninterference with a simpler computational model, where only initial and final memories are modeled—not interactions between users and system.

6.1.2 Escape Hatches

Escape hatches are trusted processes at the level of program expressions. For Sabelfeld and Myers (2003a), an escape hatch *declassify*(e) indicates that the result of evaluating *declassified expression* e may be considered to have low confidentiality. So, high confidentiality information stored in variables of expression e may flow to the low-confidentiality final evaluation of e, signifying a what-declassification. In the general case, an escape hatch also involves a *source label* ℓ and a *target label* ℓ'. The source label signifies the maximum sensitivity that this escape hatch can declassify data from; the target label signifies the minimum sensitivity that this escape hatch can declassify data to. For simplicity, we consider only two labels (i.e., high and low), in which case an escape hatch simply declassifies data from one label to the lower one.

An appropriate information flow policy would state that high confidentiality information is allowed to flow to entities with low confidentiality only through escape hatches. Sabelfeld and Myers (2003a) proposed *delimited release* as security condition that captures this intuition.[1]

[1]The original definition of delimited release can handle an arbitrary lattice of

Definition 6.1 (Delimited Release)**.** Given a mapping Γ from variables to labels in $\{L, H\}$ and a deterministic program C containing exactly n escape hatches $\mathit{declassify}(e_1), \ldots, \mathit{declassify}(e_n)$, program C *satisfies delimited release for* $\langle \{L, H\}, \sqsubseteq \rangle$ iff the following holds:

$$\forall \tau_1, \tau_2 \in \mathcal{T}_C^{\text{fin}} : \tau_1.M_0 =_L \tau_2.M_0 \ \wedge \ \tau_1.M_0 =_D \tau_2.M_0 \\ \implies \ \tau_1.M_{\downarrow} =_L \tau_2.M_{\downarrow}$$

where

$$M =_L M' \ \triangleq \ \forall x \in \mathit{var}(C) : \Gamma(x) = L \implies M(x) = M'(x) \quad \text{and}$$
$$M =_D M' \ \triangleq \ \forall i \in [1, n] : \ M(e_i) = M'(e_i)$$

Notice that delimited release extends ONI with a requirement $=_D$ on memories. Here, the corresponding initial memories need to agree on values of escape hatches in addition to values of low variables. Under this agreement, any variation on the rest of the memory components should not be propagated to the low variables at the final memories.

An implication of delimited release is that a declassified expression e is considered declassified from the very beginning of each execution of command C, independently of where escape hatch $\mathit{declassify}(e)$ is actually found in C. Askarov and Sabelfeld (2007b) illustrate this point using the following example:

$$l := h; \ C'; \ l := \mathit{declassify}(h) \tag{6.1}$$

Command (6.1) satisfies delimited release, even though high variable h is directly assigned to low variable l. This is because h appears later in an escape hatch, and thus h is considered declassified (by delimited release) from the beginning of the program. Under *localized delimited release* (Askarov and Sabelfeld, 2007b), an expression in an escape hatch is considered declassified only after the evaluation of that escape hatch. So, command (6.1) does not satisfy localized delimited release. Later, Askarov and Sabelfeld (2009) express localized delimited release using knowledge-based semantics.[2] Notice that localized delimited release can specify not only what information may be declassified, but also where

labels, whereas this definition, for simplicity, handles only a two-level lattice.

[2]Section 6.3 discusses knowledge-based semantics for declassification.

in the program code and when in the execution this declassification may occur.

In general, when proposing an information flow policy that handles escape hatches, there are at least three design decisions that need to be made. The first is whether this policy is sensitive to the location of escape hatches in a program (e.g., delimited release is not sensitive, localized delimited release is sensitive). The second decision is whether the effect of an escape hatch is permanent or not. For example, after the evaluation of an escape hatch $\mathit{declassify}(e)$, for how long can e be considered declassified and thus be safely revealed to the public by subsequent commands? The *resetting approach* (named by Broberg and Sands (2009)) has declassifications last until the next declassification occurs.[3] But for delimited release, a declassification lasts until the end of the execution.

The third design decision considers the value that is expected to be declassified by an escape hatch. Specifically, given an escape hatch $\mathit{declassify}(e)$, both delimited release and localized delimited release consider the value of e at the initial state of execution as the information allowed to be declassified—as opposed to the value of e at the program state when the escape hatch is executed. Consider, for example, the following program:

$$\mathit{avg} := \frac{h_1 + h_2 + h_3}{3};\ l := \mathit{declassify}(\mathit{avg}) \tag{6.2}$$

where h_1, h_2, h_3, and avg are high confidentiality variables, while l is a low confidentiality variable. If the average of h_1, h_2, h_3 is intended to be declassified, then program (6.2) would be considered secured. However, (6.2) does not satisfy neither delimited release nor localized delimited release. This is because these security conditions allow only the initial value of avg (i.e., the argument of escape hatch $\mathit{declassify}(\mathit{avg})$) to be declassified—not the value of avg after the first assignment. So, for these security conditions, the reference point of the declassification is always the initial state. Lux and Mantel (2009) increase the expressiveness (and complexity) of these security conditions to support declassifications with reference points anywhere in the code.

[3] The resetting approach was not originally proposed to handle escape hatches, but its main idea can be naturally applied to escape hatches.

Several extensions of delimited release have been proposed. *Composite delimited release* (Magazinius *et al.*, 2010) has been developed for *mashups*, which are web applications that combine content from multiple domains. Here all domains should agree on the initial value of an expression to be declassified.

The security conditions above specify information permitted to be released (e.g., flow from high to low). However, there are cases where a system is obliged to release information, too. Chong (2010) combines delimited release with *required release* to define *bounded release*, which stipulates both what information may be released and what information must be released, ultimately setting upper and lower bounds of the released information. Contrary to other security conditions discussed so far, the author points out that required release is instantiated as a trace property.

Based on relational logic, Chudnov *et al.* (2014) use pre- and post-conditions in program code to implement escape hatches. In particular, the authors propose the use of **assume** and **assert** commands in the program code as a way to express (and enforce) information flow policies for reclassification. For example, given a program that checks password equality and declassifies the result, "**assume** $\mathbb{A}$ ($public\_in = secret\_in$)" should be inserted at the beginning of the code and "**assert** $\mathbb{A}$ $result\_out$" should be inserted at the end, meaning that if two executions agree (i.e., $\mathbb{A}$) on whether $public\_in = secret\_in$ holds, then they should also agree on the result of the check. The security condition that the authors present stipulates that a trace is secure if any other trace, either satisfies the **assert** clause (i.e., agreement $\mathbb{A}$ with respect to the first trace), or fails to satisfy the **assume** clause, or diverges. Chudnov and Naumann (2018) extend the above framework to handle inputs and outputs during execution.

6.1.3 Functions

Some researchers have aimed to make information flow policies separate from and independent of the program code. An information flow policy that is simply represented by a lattice of labels (noninterference instantiations, such as ONI, capture the semantics of such policies)

enjoys this independence, since its meaning does not depend on the specifics of a particular program. On the other hand, many definitions of delimited release are code-dependent, because they rely on annotating the program (i.e., *declassify*(e)) to identify escape hatches and the code location where those escape hatches can be used.

Li and Zdancewic (2005a) propose declassification policies that are code-independent. Here a confidentiality label is a set of functions (specifically λ-terms) through which the associated program inputs may be declassified from secret to public. *Relaxed noninterference* is then proposed as the security condition that captures the semantics of these labels: the program can be rewritten in a form where only specified functions are applied to secret inputs to compute public outputs.

Li and Zdancewic (2005b) then extend the framework above to support integrity labels, which are sets of functions, too. Dual to confidentiality labels, which are associated with program inputs, integrity labels are associated with program outputs. For example, the integrity label $\{\lambda x.x\%10\}$ associated with an output stipulates that this output could be computed by first applying $\lambda x.x\%10$ to (possibly) low integrity inputs and then performing additional computation on the result. So, by applying $\lambda x.x\%10$ to (possibly) low integrity inputs, the output may be considered of high integrity, and thus $\lambda x.x\%10$ may cause an endorsement. Relaxed noninterference is then extended to support these integrity labels: the program can be rewritten to a form where the result is computed using one of the functions in the integrity label.

The challenging aspect of the above two approaches, though, is enforcement: one needs to (conservatively) decide whether the applied computation implemented in code is equivalent to any function in the label associated with the argument, in order to deduce whether a reclassification should be performed. This challenge can be alleviated when considering a simplified framework. Kaneko and Kobayashi (2008), for instance, assign unique identifiers to functions that may cause declassifications, and then use these identifiers in labels and programs to refer to these functions. The enforcement problem is then reduced from function equivalence to identifier equivalence. Of course, the downside is an increased dependency of policy on code. The framework proposed by Kaneko and Kabayashi can specify how many times a function

(identifier) should be applied for a declassification to be triggered. The authors introduce *linear relaxed noninterference* as an extension to relaxed noninterference.

In the context of object oriented programming, instead of considering a set of arbitrary functions as a label, Cruz *et al.* (2017) consider an object interface as a label, with the idea that an object interface dictates the set of methods through which information about the object can flow to an observer. Thus, depending on the sensitivity of data that the observer is allowed to read, such an interface might limit the view of the object. Similarly, Hicks *et al.* (2006) enable a principal to declare *trusted methods* to declassify data. The authors propose *noninterference modulo trusted methods* as the desired security condition.

By abstracting functions with identifiers, Kozyri and Schneider (2020) present a generalized framework for expressing reclassifications. Here, a *reactive information flow* (RIF) label associated with a value v maps each sequence of function-identifiers applied to v, to restrictions on how the resulting value v' may be used. Because restrictions imposed on v' might be different from those imposed on v, a RIF label can express arbitrary reclassifications that are triggered by function applications. As an example, a RIF label on some training data can specify that this data may initially be considered public, but once used to train a proprietary machine learning model, the resulting model should be considered secret. Here, information from public inputs (i.e., training data) is required to flow to a secret result (i.e., proprietary model), which constitutes a classification. The authors propose two special classes of RIF labels. *RIF automata* are automata-based structures whose transitions correspond to function-identifiers and states correspond to restrictions. *κ-labels* are stack-based structures specialized to specify how confidentiality restrictions change when consecutive cryptographic operations are applied to value. *Piecewise noninterference* (PWNI) is proposed as a security condition that handles reclassifications specified by RIF labels.

Cryptographic operations are functions of special interest. Here, an information flow policy would allow secret data to flow to the public only if this is data is encrypted. *Computational indistinguishability* (Laud, 2001) is a security condition that captures this intuition. It requires

that polynomial-time bounded adversaries cannot efficiently retrieve secret information from ciphertexts. Notice that noninterference (e.g., ONI) is not satisfied by programs that use encryption, because there is information flowing from the plaintext to the ciphertext, though not enough to efficiently recover the plaintext. For arbitrary cryptographic primitives (e.g., encryption, authentication, pseudo-number generators) in a reactive computational model, *computational probabilistic noninterference* (Backes and Pfitzmann, 2002) captures the intuition that polynomial-time bounded adversaries can achieve their goals (e.g., learn something secret or influence something trusted) with only negligible probability.

6.1.4 Execution State

A flow of information from one entity to another may be allowed depending on the execution state of the system. *Targeted conditional delimited release* (Do *et al.*, 2016) specifies not only *what* information is allowed to be released by escape hatches, but also *when* this release may occur. For example, such a policy can specify that the majority of secret votes (i.e., escape hatch) may be declassified at 8 p.m. (i.e., condition on execution state).

Broberg and Sands (2009) propose *flow locks* to specify declassifications based on program states. Here, a label p is a set of clauses of the form $\Sigma \Rightarrow a$, which specifies the conditions Σ under which the associated data is allowed to flow to entity a. In particular, Σ is the set of *locks* that should be open for the associated data to flow to a. For example, a variable h might be associated with label $\{high;\ Decl \Rightarrow low\}$, which means that information from h may always flow to *high*, and it may flow to *low* only when lock *Decl* is open in the execution state. The following program satisfies this label:

$$open\ Decl;\ l := h;\ close\ Decl \tag{6.3}$$

where "*open Decl*" opens lock *Decl* and "*close Decl*" closes lock *Decl*. The authors propose a termination-sensitive and termination-insensitive version of a security condition that formally characterize the restrictions imposed by flow locks.

Broberg and Sands (2010) later proposed *paralocks*, which increase the expressiveness of flow locks by incorporating principles from role-based access control. Here is an example label:

$$\{\forall x.\{Bidder(x), AuctionClosed\} \Rightarrow x\} \tag{6.4}$$

which specifies that the associated data may flow to any principal x that is a *Bidder*, provided lock *AuctionClosed* is open in the execution state.

Predicates on execution state have also been employed by Chong and Myers (2008) to specify information flow policies. In addition to declassification, the proposed policies can also express erasure. For example, such a policy can express that user data may flow to the web application until the session ends, at which point this user data (and any other information computed based on the user data) should be erased from the system. The authors propose *noninterference according to policy* to capture the semantics of labels. The enforcement of this security condition involves both static analysis of the program and a runtime mechanism that ensures values are actually erased when the specified predicate on execution state is satisfied. By restricting predicates on execution state to reference only the program counter (i.e., program point in the code), instead of referencing arbitrary variables in the program, Hunt and Sands (2008) avoid the use of such a runtime mechanism for enforcing the proposed security condition, *end-to-end erasure*.

Information flow policies that specify reclassifications based on execution state tend to capture a temporal aspect of how restrictions on data should change. For instance, such policies might specify when a variable should be erased, or for how long a variable may be considered public. So, temporal logics seem to be suitable for expressing such information flow policies. Dimitrova *et al.* (2012) and Clarkson *et al.* (2014) follow this intuition and propose appropriate extension of linear-time temporal logic (LTL). Balliu *et al.* (2011) employ temporal epistemic logic to express information flow policies for what, where, and when declassification.

The policies discussed so far in this subsection are known as *dynamic policies*, where the set of allowed and forbidden flows can change during

execution. Broberg *et al.* (2015) survey dynamic policies. The authors describe a dynamic policy as "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". They propose a characterization of dynamic policies in terms of five *facets*[4] and in terms of a three-level *hierarchy of control*. *Level 0* is a set of possible flow relations between information sources (e.g., input variables) and sinks (e.g., output channels). A flow relation indicates that information from the source is allowed to flow to the sink. *Level 1* selects which flow relations are allowed. *Level 2* constitutes a meta policy for controlling the way in which the current flow relations (Level 1 control) may be changed.

More recently, Li and Zhang (2021) systematized previously proposed dynamic policies using a framework based on knowledge-based semantics (see Section 6.3). Interestingly, this framework can express both *transient* and *persistent* policies: "A dynamic security policy is persistent if it always allows to reveal information that has been revealed in the past. Otherwise, the policy is transient". Consider for example a secret s that is declassified to the public and, later, erased again. A persistent policy would allow s to flow to the public even after the erasure, because this information has been already revealed to the public. However, a transient policy would not allow s to keep flowing to the public after the erasure.

The dynamic policies discussed in this subsection can express "when" (and "where") information may be reclassified based on execution-state predicates. These policies can also express "what" information may be declassified at the level of variables. For example, they can specify for the content of secret variable s (i.e., s instantiates the "what" dimension) to be declassified at midnight (i.e., midnight instantiates the "when" dimension). However, these policies cannot yet express reclassifications at the level of expressions: they cannot express that only expression "s *mod* 3" of secret s may be declassified at midnight. Expression-level reclassification can be specified by policies discussed in the previous

[4] *Termination sensitivity*, *time-transitive flows*, *replaying flows*, *direct release*, and *whitelisting flows*.

subsection, though they do not support the "when" dimension, in terms of arbitrary execution-state predicates. A unifying framework for expressing "when" (i.e., condition on execution state) and "what" information (i.e., expression of variables) may be reclassified is missing from the literature.

6.1.5 Interactions

When one considers event-driven programs (e.g., JavaScript web applications), events (e.g., MouseClick, KeyPress) are state components of special interest that may be declassified. Vanhoef *et al.* (2014) propose policies that may declassify the result of applying a function D to a confidential event sequence. An example of such policy would specify: "the average of mouse click coordinates can be declassified after 100 clicks". The authors propose *noninterference under* D, which stipulates that if two input event sequences are indistinguishable to low principals and yield the same result when D is applied, then the output sequences should be indistinguishable to these low principals, too.

Alternatively, the conditions under which declassification may occur can be specified in terms of events. Micinski *et al.* (2015) employ linear-time temporal logic (LTL) formulas to specify conditions on event sequences under which sensitive information about inputs may be released. Here a *declassification condition* is a formula $\phi \triangleright S$, meaning that if LTL formula ϕ over events holds at the time an input occurs, then that input is declassified to label S. The authors propose *interaction-based noninterference* (IBNI) as the intended security condition: observational determinism should hold after all inputs have been declassified according to the declassification conditions.

Researchers have studied reclassification policies based on the interactions among components of a distributed system. Greiner and Grahl (2016) focus on *component-based systems*, where components are programs that interact with each other through messages. The authors show that a noninterference statement for what-declassification is compositional for sequential and parallel compositions that satisfy certain restrictions.

Guttman and Rowe (2015) study declassification at the level of a distributed system, too. Here a distributed system is represented by a directed graph, where each node encloses a set of local behaviors and each edge is a channel for synchronous transmission of messages. Set *src* of nodes provide the system with possibly sensitive information and set *obs* of nodes are those that produce observations to the environment. The authors show that the sensitive information revealed (or declassified) to nodes in *obs* is always upper-bounded by the sensitive information collectively revealed to the nodes that form the cut of that graph. This means that nodes in *obs* cannot learn more information about sensitive data in *src*, that what is learned by nodes in that cut.

Best and Darondeau (2012) express declassification policies for *Petri nets*, which can model distributed systems. A Petri net is a directed graph, where a node represents a *place* or a *transition*, an edge from a place to a transition dictates that place to be an input to the transition, and an edge from a transition to a place dictates that place to be an output of that transition. In terms of expressiveness, Petri nets lie between finite state automata and Turing machines. When labeling transitions with labels in $\{High, Low, Decl\}$, the authors express intransitive noninterference as the requirement that a Petri net where all transitions labeled *Decl* are removed is language equivalent to the Petri net where all transitions labeled *Decl* or *High* are removed.

6.2 Robustness

When an operation in a program is allowed to declassify information, one might need to ensure that an active adversary is not able to exploit this operation to leak information that should not be declassified. *Robust declassification* (Zdancewic and Myers, 2001) captures this intuition formally. Here an active attacker can observe and modify the behavior of a system (e.g., overwrite parts of memory). Robust declassification stipulates that active attackers should not learn any additional confidential information through active attacks than what they could have learned through only passive observation. Notice that the fundamental idea of robustness is not confined to reclassifications. In general, robustness can be regarded as a meta-policy that stipulates the following:

Given an attacked C and a non-attacked C' version of a program, C and C' should exhibit equivalent behaviors with respect to a particular criterion, such as satisfying a certain policy.

Myers *et al.* (2006) generalize robust declassification by allowing untrusted code and data to be part of the system. To do this, integrity labels are associated both with data and program code. If data or code is provided or affected by the attacker, then it will be associated with a label that represents low integrity, otherwise it will be associated with a label that represents high integrity. The code provided by the attacker cannot be arbitrary; it should not violate confidentiality and integrity restrictions directly, say, by assigning high confidentiality variables to low confidentiality variables. So, the authors consider *fair attacks* A, which is code provided by the attacker that can read and write only low confidentiality and low integrity variables.

Robust declassification is then formalized relative to fairs attacks. If $C[A]$ denotes a program C where the low integrity components are provided as an attack A, then robust declassification stipulates: whenever the behavior of program $C[A]$ is indistinguishable on some memories, any change of the attacker's code to any other attack A' still will leave the behavior of program $C[A']$ indistinguishable on these memories. So, the attacker does not learn anything new by observing the behaviors of $C[A']$, comparing to what they already know from observing the behaviors of $C[A]$. Notice that a set of at least four execution traces of the system is needed to establish that this system violates robust declassification: two traces of $C[A]$ that are indistinguishable for an attacker on two initial memories and two traces of $C[A']$ that are not indistinguishable on these memories. So, robust declassification is a 4-safety hyperproperty.

Myers *et al.* (2006) employ the following program C as an example that does not satisfy robust declassification:

$$\textbf{if } u > 0 \textbf{ then } l := \mathit{declassify}(h) \textbf{ else } l := 2$$

where u is an untrusted variable written by the adversary, l is a low confidentiality variable read by the adversary, and h is a high confidentiality variable. Here, the adversary can control whether h will be declassified, by choosing appropriate values for variable u. Specifically,

when the adversary chooses u to be 1 and executes the resulting program $C[u = 1]$ on two indistinguishable memories M and M', which disagree on the value of high variable h but agree on all low variables, then the adversary reads distinguishable values at l. However, when the adversary chooses u to be 0 and executes the resulting program $C[u = 0]$ on these indistinguishable memories M and M', then the adversary reads always the same value 2 at variable l. This discrepancy between the behavior of $C[u = 1]$ and $C[u = 0]$ is the reason why C does not satisfy robust declassification.

Myers *et al.* (2006) also propose *qualified robustness*, which can allow untrusted code to control declassification though the endorsement of untrusted data (i.e., provided by the attacker). Later, Askarov and Myers (2010) propose a more accurate semantic definition of qualified robustness.

Cecchetti *et al.* (2017) present *transparent endorsement* as the dual of robust declassification. Transparent endorsement forbids endorsement of information that cannot be viewed, so it uses confidentiality to limit relaxations of integrity. The authors propose *nonmalleability* as the combination of robust declassification and transparent endorsement.

6.3 Knowledge-based Semantics

An information flow policy for confidentiality could be expressed in terms of allowed or forbidden knowledge acquisition. This approach is based on the following premise: an entity flows to a principal if and only if this principal learns new information about that entity. So, by restricting knowledge acquisition, one restricts the flow of information. Knowledge-based semantics is particularly compelling for expressing reclassifications, by intuitively specifying how the knowledge of adversaries about certain data is allowed to change during system execution. Though researchers initially employed knowledge-based semantics to reinterpret noninterference.

Epistemic logic has been employed to express information flow policies. An epistemic logic can reason about knowledge, which is semantically modeled by *Kripke structures*: a principal *knows* a predicate Φ if at all worlds that are considered possible by this principal, Φ holds.

Halpern and O'Neill (2008) employ epistemic logic to express a wide range of security conditions, including generalized noninterference and probabilistic noninterference.

Askarov *et al.* (2008) express TSNI and TINI using a simpler semantic model of knowledge. From the execution traces generated by this system, principals may observe sequences of outputs. By observing a certain sequence of outputs, a principal accumulates knowledge about initial secrets. And this knowledge is modeled by the set of possible initial memories that would have led to that sequence of outputs. Knowledge is monotonic: by making more observations, a principal's knowledge about initial secrets increases (i.e., the set of possible initial memories becomes smaller) or remains the same, but never decreases. Under the proposed model of knowledge, the authors express TSNI as: "at each step of output (real output or divergence signal) the attacker learns nothing new about the initial high memory". To express TINI, they allow knowledge to increase only through progress (i.e., that an output appeared). The authors prove equivalence between a knowledge-based statement of TINI and a trace-based one (e.g., ONI). Building on the model of knowledge proposed by Askarov *et al.* (2008), Balliu (2013) defines a logic with temporal and epistemic operators. This logic can be used to express several security conditions, including generalized noninterference.

Knowledge-based semantics has been employed to express information flow policies for reclassifications. Askarov and Sabelfeld (2007a) propose *gradual release*, which specifies that the adversary's knowledge about initial secrets may increase only after the occurrence of release events (e.g., execution of *declassify*(e)). *Conditional gradual release* (CGR) (Banerjee *et al.*, 2008) bounds by how much the adversary's knowledge may be increased, based on *flowspecs* that proscribe information allowed to be released. Rocha *et al.* (2010) build on the ideas of CGR and introduce *policy graphs*, which dictate the path of operators a secret value should flow through in order to be declassified. The authors define *policy controlled release* (PCR) as the policy scheme that specifies allowed flows of information according to policy graphs.

Whether the knowledge of an adversary about an initial secret changes might depend on the adversary's ability to recall past events. Askarov and Chong (2012) apply this observation to systems that

support dynamic policies. The authors express an information flow policy with respect to an adversary that is modeled as a state-based machine, updating its state according to the observed events. The more powerful a state-based machine is, the more information can be recalled by the adversary, leading to a stronger adversary. The authors show counterintuitive scenarios where a program might be insecure for a certain adversary, but secure for a stronger adversary. This result argues for defining knowledge-based security policies in terms of change of knowledge, rather than absolute bounds on knowledge.

In fact, the framework proposed by Askarov and Chong (2012) can give a knowledge-based interpretation for erasure: a piece of information is erased if and only if an adversary that cannot recall any knowledge before the occurrence of the erasure does not learn anything about that information during the remaining execution. This line of research supports erasures only for program inputs, but they cannot express erasures for intermediately computed values (e.g., values that should be considered secret, even though they have been computed based on public information). And approaches that do support erasures for intermediately computed values (e.g., Kozyri and Schneider, 2020), employ program transformations that lead to more restrictive policies than desired. So proposing security conditions that specify erasures for arbitrary computed values, without introducing unnecessary restrictiveness, is still an open problem.

Glasgow and MacEwen (1989) present *obligation logic* as the analogous of an epistemic logic for integrity. The authors diverge from the intuition we have built so far about integrity restrictions (see Section 3.1), which aim to prevent the contamination of trusted information with untrusted information. They instead equate integrity policy with the availability requirement that procedures should obtain inputs from certain sources: "integrity is a requirement that certain information flows take place between procedures". So, their security condition for integrity is actually interpreted as a liveness hyperproperty: information will eventually flow, and thus new information will be eventually known. The authors define an obligation logic by extending an epistemic logic with a modality for obligation. For example, an information flow property for integrity would say: "for certain pair of procedures i and j, if i knows ϕ then j is obligated to know ϕ".

7

Information Flow Policies and Authorization

Authorization policies restrict who can perform what action on which entity. *Access control* (Lampson, 1973) and *capability-based* frameworks are examples of authorization techniques that can specify, for instance, which users may read or write certain files in a system. Researchers are interested in understanding how information flow policies, and authorization policies are related, given that they both impose restrictions on data. It can be argued that authorization policies and information flow policies are complementary. For example, authorization policies can be deployed by information flow policies to control the occurrence of reclassifications; information flow policies can be deployed to specify that authorization decisions should not leak sensitive data or depend on untrustworthy information.

7.1 Information Flow Policies versus Access Control Policies

An *access control policy* specifies whether a given principal can perform a given action on a given object. An access control policy is interpreted as a trace property, since examining single executions is enough to decide whether that policy is satisfied.

Given that information flow policies are interpreted as hyperproperties and access control policies are interpreted as trace properties, information flow policies are, in general, more expressive than access control policies. So, in principle, an information flow policy could express a given access control policy, though not always as naturally. For example, the access control policy "s may be read only by Alice", could be expressed by the information flow policy "s may initially flow only to Alice through a read access, and the returned value of that read access may then flow to everyone".

In general, access control policies cannot precisely express arbitrary information flow policies, but access control policies can enforce (that is, conservatively approximate) information flow policies (Rushby, 1992; van der Meyden, 2007). Enforcement is possible because access control policies can prevent flows between objects and principals (Bell and LaPadula, 1973): if information should not flow from Alice to Bob, then there should be no object that Alice can alter and Bob can read. Similarly, if information should not flow from object o to object o', then there should be no principal that can read o and alter o'.

Conversely, access control policies can imply information flow policies: for example, if Alice is allowed to read object o and write object o', then information is implicitly allowed to flow from o to o' If access control rights may be transferred between principals by delegation, then additional flows are implicitly specified (Bishop and Snyder, 1979) between objects. Jaume (2012) uses these ideas to translate access control policies into information flow policies and then compare the resulting policies to familiar information flow policies in terms of expressive power.

McLean (1990b) succinctly compares access control policies to information flow policies:

> [Access control policies] concern themselves with particular controls over files in the computer rather than limiting themselves to the relation between input and output, making it harder to reason about requirements and prejudicing the programmer against alternative, possibly better, implementations by creating a mindset in which alternatives are

> not considered. A more abstract specification would ignore system internals and deal directly with input/output relationships. It could require, for example, that output to low-security users not allow those users to infer properties of higher-level users' input.

Nevertheless, as argued by Stoughton (1981), both access control policies and information flow policies are needed for a secure system. For example, as described earlier in this section, an information flow policy that can capture a high-level guarantee for the system, could be interpreted as concrete and simple access control policies that can then be enforced on the system. Nanevski *et al.* (2013) propose *Relational Hoare Type Theory* to be a single framework that can express both access control and information flow policies for a system.

7.2 Authorization for Information Flow Policies

Information flow policies can use authorization decisions to specify whether certain reclassifications are permitted. For example, an information flow policy may allow a file to be declassified if it is performed by an authorized principal. The *Decentralized Label Model* (DLM) (Myers and Liskov, 1997) was the first framework to increase the expressiveness of information flow policies with authorization decisions. For confidentiality, a DLM label associated with some data specifies the owner of that data and the set of principals—called *readers*—to whom this data may flow. Data may then be declassified, adding more principals to the readers set. Such a declassification may be performed only by program code that runs under the *authority* of a user *trusted* by the owner of that data.

DLM labels can specify *who* is allowed to declassify information, based on a centralized trust hierarchy between principals. Put in terms of authorization, a DLM label allows a run-time authority (i.e., a principal) to increase (i.e., perform an action on) the readers set in the label (i.e., object) of declassified values, provided that authority is trusted by the owner in that label. Essentially the owner of data is also considered the owner of the associated label (Chen and Chong, 2004), since this

owner decides how the label changes (by adding more principals in the readers set). In general, a DLM label consists of a confidentiality and an integrity component, and the set of DLM labels forms a lattice.

DLM can express and enforce declassifications within decentralized systems of mutual distrust, where there is no centralized authority that determines which computation is trusted to declassify sensitive information. Chong and Myers (2006) point out that in such a system, where mutually distrusting principals interact, who is considered to be the adversary differs from one principal to another. So, the authors extend robust declassification and qualified robustness to express information flow restrictions against all possible adversaries, and they show that DLM satisfies these extensions. Pedersen *et al.* (2015) extend DLM with *clock expressions* that additionally specify when information may be released to a reader. The resulting *Timed Decentralized Label Model* (TDLM), whose underlying semantics is given based on a network of timed automata, is suitable for Internet-of-Things applications.

Drawing inspiration from capability-based frameworks, Stefan *et al.* (2011a) propose *Disjunction Category (DC) labels* to specify information flow policies within a system of mutually distrusting parties. A DC label consists of two Boolean formulas over principals (in conjunctive normal form and without negations): one for confidentiality that represents principals allowed to observe the associated data, and one for integrity that represents principals allowed to modify the associated data. DC labels form a lattice, where the flow relation is based on logical implication. Declassification and endorsement may be performed by code running on behalf of principals possessing appropriate *decentralized privileges*. These privileges, which function as capabilities, are Boolean formulas over principals, too. So, these privileges may be used to strengthen the assumption of logical implications, and thus, allow flows that were previously forbidden. *LIO* (Stefan *et al.*, 2011b), which is an information flow control system in Haskell, employs DC labels.

Montagu *et al.* (2013) present *label algebra* as a unifying mathematical framework in which a superset of the label structures presented above (DLM and DC labels) can be expressed and compared against each other. Such a label algebra consists of a relation on labels and a relation on authorities. This relation on authorities directly captures

the trust hierarchy of DLM and can also encode the logical implication between the decentralized privileges for the DC labels.

Capabilities have been also employed by the operating system *Flume* (Krohn *et al.*, 2007) to support information flow policies for decentralized systems. In Flume, labels are sets of tags, and thus, they form a lattice under the partial order of the subset relation. Each process is accompanied by privileges, which are modeled as capabilities, to add or remove tags from labels of input data, causing reclassifications. Krohn and Tromer (2009) use CSP to formalize the security condition that Flume enforces, which is based on noninterference. Other information flow control systems that are based on capabilities include *Laminar* (Roy *et al.*, 2009), *HiStar* (Zeldovich *et al.*, 2006), *Asbestos* (Efstathopoulos *et al.*, 2005), and *Aeolus* (Cheng *et al.*, 2012).

7.3 Information Flow Principles for Authorization

An authorization mechanism grants or rejects an attempted access based on a given authorization policy. But such a policy might be sensitive. So, deciding whether an access is granted might reveal this sensitive information. For example, an authorization policy might consist of (i) a list of suspects and (ii) an assertion saying that only a non-suspect is allowed to read document X. By requesting a read access to X and then observing whether that access is granted, users could deduce whether they are suspects, which is presumably sensitive information. Becker (2010) addresses this problem by associating authorization policies with sensitivity labels and using two security conditions, noninterference and *opacity*, to proscribe leaks of sensitive information through authorization decisions. Opacity (Bryans *et al.*, 2008) is equivalent to opaqueness, which is discussed in Section 4.6, and it was originally applied to cryptographic protocols. Becker (2010) argues that, in this context, opacity seems to be more expressive than noninterference.

Components of an authorization policy might encode sensitive information, because those might be modified by programs that process sensitive inputs. Consider, for example, the trust hierarchy between principals (i.e., an authorization mechanism), which is used by DLM (Myers and Liskov, 1997) to determine allowed declassifications and endorse-

ments. If this trust hierarchy is dynamically modified by program code that adds or removes trust relationships between principals, then sensitive information might flow from program inputs to this hierarchy. When the enforcement mechanism decides to grant or deny a declassification to the public, based on the modified trust hierarchy, the encoded sensitive information might then be leaked to the public observers: by observing whether a value is declassified, one can deduce information about the modified hierarchy, and possibly about the sensitive inputs that contributed to that modification. So, in this example, information flows from sensitive program inputs, to the trust hierarchy, and then to observers of the system's behavior.

An information flow policy can be used to specify which of the flows between computational components (e.g., program implementing desirable functionality) and enforcement components (e.g., the trust hierarchy) are allowed, avoiding unintentional leaks to the public. *FLAM* (Arden *et al.*, 2015) is a framework that reasons, in particular, about the flows between code and the trust hierarchy. With FLAM, different observers might be allowed to observe the effects of different parts of the trust hierarchy. FLAM also models the possibility that the trust hierarchy is stored in a distributed way, so that the communication among distributed nodes that is needed to answer authorization queries also could leak information.

Another instance of restricting information flow between computational and enforcement components is addressed by the security-typed language *RX* (Swamy *et al.*, 2006). For authorization, RX supports role-based security policies, where a role identifies a set of principals. Programs are then able to dynamically view and modify this set, by adding or removing principals. So, the roles might encode sensitive information, which might then leak via observable enforcement actions that depend on these roles. To prevent those leaks, RX employs *metapolicies* to capture the sensitivity of information encoded in these roles. RX is then shown to satisfy an information flow policy based on noninterference.

In general, an enforcement mechanism might introduce additional enforcement components (e.g., metapolicies) to capture and protect information that might have influenced existing enforcement components (e.g., role-based security policies). But the additional enforcement

components might encode sensitive information, too. So, even more enforcement components might be needed. Kozyri *et al.* (2019) follow this argument to its limits, for the case of information flow control, by considering *chains* of labels (i.e., enforcement components) and proposing appropriate security conditions for the overall system. Zheng and Myers (2007) give an information flow policy for programs that explicitly declare and manipulate chains of labels.

Information flow policies have been employed to safeguard the integrity of authorization mechanisms, too. For example, FLAM does not allow actions taken and data provided by adversaries to influence the trust hierarchy. Thus, adversaries cannot drive the system to perform unintended reclassifications of sensitive data. As another example, consider a capability-based system where delegation is performed by propagating capabilities from one party to another. To safeguard this delegation, Dimoulas *et al.* (2014) employ a security condition based on noninterference to specify that only trusted parties may influence the use and propagation of capabilities.

8

Quantitative Information Flow Properties

For a system that processes sensitive information, it might be infeasible or impractical to demand no leakage of sensitive information to the public. Section 4 discusses how sensitive information might be inadvertently leaked through covert channels (e.g., execution time, heat-emission) that are created during the system execution. Sections 6 and 7 argue that there are conditions where a leak is actually desirable. But for a leak that is unavoidable or desirable, one might be interested in understanding its magnitude, to be convinced that it does not cause more harm than anticipated. So, there is a need to quantify the information that is leaked. A wide variety of leakage measurements have been studied within the research area of *quantitative information flow* (QIF), whose foundations are built on *information theory*. The book of Alvim *et al.* (2020) offers a deep understanding of QIF, including program analyses for measuring information leakage. This section focuses on leakage measurements that could be employed to express information flow properties.

Quantitative information flow properties set bounds on the amount of information allowed to flow between entities. So, the amount of information can be considered as a quantitative condition for characterizing allowed or forbidden flows: a flow is allowed only if it conveys an amount

of information below a certain threshold. An example of a quantitative information flow policy associated with a secret would specify that at most one bit of information is allowed to leak from that secret to the public. Previous sections have presented information flow policies and properties that use qualitative conditions (e.g., labels, state predicates, applied operations, authority) to allow flows between entities.

8.1 Expressing Policies

QIF is mainly employed to protect the confidentiality of data. Here, a secret X is accompanied by a *prior distribution* π, where $\pi(X = x)$ signifies the probability for X to have value x. Given a system that takes secret X as an input, prior distribution π models the knowledge that an adversary has about X prior to the execution of that system. If this system produces an output Y, then by observing the value of Y, during or at the end of system execution, the adversary might update their knowledge about X (e.g., some values of X might be less or more likely, given the specific output value). This updated knowledge of the adversary is now modeled by a *posterior* distribution on X. A leakage measurement quantifies the difference between the prior and posterior distribution, and thus, it captures the quantity of information leaked from X to Y. Alvim *et al.* (2012b) propose a parameterized measurement of leakage that can express a class of known leakage measurements.

An information flow property can be expressed as a bound on a leakage measurement. For instance, an information flow property could set bounds on the *capacity* of a system, which is the maximum amount of information that this system can leak under a given measurement over all possible prior distributions. As another example, *robust* leakage measurements (Alvim *et al.*, 2014a), which place leakage bounds independent of the prior distribution, could also be used to express information flow properties. Bounds on the *rate of leakage* (Malacaria, 2007), for programs with **while**-loops, can form information flow properties, too.

Quantitative information flow properties for integrity could be expressed as bounds on integrity measurements proposed by Clarkson and Schneider (2010b): *contamination* and *suppression*. Contamination

measures how much untrusted inputs have influenced trusted outputs, and it is dual to leakage. Suppression is based on program correctness. Consider a specification and a program that is intended to implement that specification. An output produced by this program might convey only partial information about the specified output. Suppression then measures how much information of the specified output is lost.

Differential privacy (Dwork, 2006) can be interpreted as an information flow property in QIF. According to Tschantz *et al.* (2020), a function applied on a database satisfies differential privacy when this function "produces almost identical distributions of outputs for any pair of possible input databases that differ in a single data point". This requirement implies that almost no information flows from a single data point to the output of the function. Researchers have followed this intuition to show that differential privacy is equivalent to setting a certain bound on a leakage measurement (e.g., Alvim *et al.* (2011)). But a bound on a leakage measurement is one way to give semantics to differential privacy. Tschantz *et al.* (2020) survey different approaches for giving such semantics, which can incorporate assumptions about the adversary's inference power, or assumptions about existing correlations between data points. The authors emphasize the interpretation of differential privacy in terms of causality and point out that the differences between the discussed approaches boils down to the distinction between correlation and causation.

Connections have been established between quantitative and qualitative information flow properties. Millen (1987) shows that if a system satisfies noninterference between the input and the output, then the capacity of this system is zero. For systems that employ cryptographic primitives, Backes (2005) shows that allowing only a negligible quantity of information flow is equivalent to computational probabilistic noninterference.

8.2 Varying the Threat and Computational Model

Different computational and threat models expose different means for information to leak from a system to an adversary. Information flow properties that are expressed based on leakage measurements adapt

accordingly to accommodate these differences. Specifically, the selection of a leakage measurement directly depends on the threat model, which describes actions that the adversary can take in an effort to correctly guess the secret. For example, according to Alvim *et al.* (2020), *Shannon entropy* is a measurement that captures the "expected number of yes/no questions to determine the secret's value". And the *Bayes vulnerability* is a measurement that captures the probability of correctly guessing the secret in one try.

A threat model also describes which entities of the system can be actually observed by an adversary. Malacaria and Chen (2008) define information leakage against a variety of adversaries: from those that observe a single low output, to those that can observe low values at every execution step. Köpf and Basin (2007), followed by Rakotonirina and Köpf (2019), consider adversaries observing side-channels (e.g., timing channels) and measure information leakage based on the number of consecutive observations on these channels. QIF has been shown to be particularly well suited for reasoning about information leakage through covert channels.

The harm caused by a leakage might depend on what was actually leaked to the adversary. Alvim *et al.* (2014b) generalize QIF measurements (e.g., Shannon entropy) to accommodate secrets that consist of multiple fields and are accompanied by a worth assignment, which maps each field of a secret to its worth. The higher the worth of a field, the more significant the harm if this field in leaked. Based on this idea, an information flow property could then describe allowed flows depending on the associated harm that they might cause.

When consecutive values of a secret are selected based on a certain strategy, then one needs to protect not only the confidentiality of the secret, but also the confidentiality of that strategy. If an adversary learns the strategy in combination with a leaked secret value, then she might be able to predict future secret values. Mardziel *et al.* (2014) consider *dynamic secrets* that change based on a strategy. For example, consider a user password that is regularly changed by appending a new digit each time. The authors propose measurements to quantify the leakage of such dynamic secrets. Later, Alvim *et al.* (2017) propose to model the adversary's prior knowledge as a distribution on strategies, which

itself is a distribution on secrets. So, the adversary's prior knowledge is modeled as a distribution on distributions on secrets, called a *hyper-distribution*. The authors argue that distributions of higher order are not necessary to soundly quantify the leakage of a secret. They also generalize QIF measurements for strategies. These measurements can then be used to construct information flow properties on strategies.

In general, what an adversary believes about the possible values of a secret might not be true. For example, an adversary might believe that it is equally likely for a binary secret to be 0 or 1, but in reality, this secret is more likely to be 0. When the adversary observes the output of such a system, she might update her belief about the secret input. Clarkson *et al.* (2005), followed by Hamadou *et al.* (2010), propose leakage measurements that are based on how the adversary's belief about a secret input changes upon the observation of an output. So, an information flow property based on these measurements would specify restrictions on how the adversary's beliefs about secrets may change.

QIF measurements have been deployed to capture flows within different computational models, such as interactive and concurrent systems. Alvim *et al.* (2012a) define for each new output the additional information that is leaked about the input sequence processed so far. Mestel (2019) show that if an interactive system is deterministic, then "the information flow is either logarithmic or linear, and there is a polynomial-time algorithm to distinguish the two cases and compute the rate of logarithmic flow". Chen and Malacaria (2007) propose leakage measurements for multi-threaded programs, considering threat models sensitive to the scheduler, the thread's timing behavior, and/or intermediate state of the computation. Bounds on these proposed leakage measurements give rise to information flow properties for the corresponding computational and threat models.

Quantitative information flow properties have been proposed for quantum system, too. Ying *et al.* (2013) recognize that even though a quantum system is a probabilistic system, probabilistic noninterference is not directly applicable to quantum systems. Instead, the authors use *quantum automata* as the computational model and they define an appropriate version of noninterference for these automata.

9

Future Directions

The literature on information flow properties offers many tools for capturing the complex restrictions on data usage required by today's digital society. Research in this field has followed technological and intellectual advances in computer science and engineering to propose properties appropriate for new computational models and threat models. At the same time, the expressiveness of information flow policies has been constantly increasing, in an effort to capture nuanced conditions under which flows are allowed or forbidden. Nonetheless, there are several open research problems, many of which are indeed fundamental. Addressing these open problems may contribute to the wider adoption of information flow policies and properties by academia and industry.

Unifying Information Flow Properties

Sections 6 and 7 include approaches that unify subsets of previously proposed information flow properties into common frameworks. A unification that is still missing though is a framework to express both state-based and expression-based conditions for reclassification of arbitrary computed values (not only input values) in an intuitive way. Complications that arise when attempting to propose such a framework

might be alleviated, if definitions for information flow are crystallized. Instead of extending noninterference with exceptions that allow for different reclassification conditions, one could express information flow properties with a set of statements of the form "if secret information flows to a public channel, then this flow should have been established because a certain condition holds". When conditions involve function applications (i.e., expressions), then such a statement would stipulate that "if secret information flows to a public channel, then this secret information should have first flowed through a certain sequence of functions". But definitions for "flow through" have not been systematized in the literature, yet.

Intelligible Policies and Labels

The inherent tension between expressiveness and simplicity is apparent in the literature of information flow policies. The more expressive the policies and labels are, the more precise the specification of allowed or forbidden flows becomes, enabling a more fine-grained control of information flow. However, increasingly expressive policies and labels become less understandable and possibly less usable. To widen the adoption of information flow policies by programmers, a balance must be struck between expressiveness and usability. Such a balance could be achieved if one first understands the needs of the industry: what patterns of policies are being employed and how frequently these patterns are needed. Then a suitable framework for information flow policies would express common patterns in an understandable and succinct way, while it would still express less common patterns by gradually increasing the complexity of the syntax. To design such a framework and align the expressiveness of proposed information flow policies with the needs of the industry, empirical user studies need to be conducted.

From User Restrictions to Information Flow Properties

Prompted by digital services, end users have to decide how their data may be used. There is an active area of research that studies appropriate user interfaces (UI) for the intuitive and informative specification of data-usage restrictions. For a system that is based on information

flow properties to achieve rigorous security guarantees, the data-usage restrictions that are chosen by the end-users should be ultimately translated into appropriate information flow properties. This seems to be an interesting research problem, where the researcher would need to propose a faithful interpretation from the UI syntax to the syntax for expressing system-wide information flow properties.

Secure Compilation

Secure compilation (Patrignani *et al.*, 2019; Abate *et al.*, 2019) ensures that the compilation of a program preserves a certain security policy. In the literature of information flow, this policy usually stipulates that the program satisfies noninterference independently of the context in which this program executes. Under secure compilation, if a source program satisfies such an information flow policy, then the target program should satisfy that policy, too. But the source program and the target program are executed under possibly different computational models, which might affect how the desirable information flow policy is expressed. So, the two different computational models might lead to two different versions of the desirable information flow policy. The question that arises is how to establish that these two versions are semantically equivalent.

Metaconditions

Robust declassification, transparent endorsement, nonmalleability, and speculative noninterference can be regarded as security metaconditions. These metaconditions complement information flow properties by imposing additional restrictions on flows, and thus, increasing the overall security of the system. As discussed earlier, robust declassification is a 4-safety hyperproperty, while many information flow properties are 2-safety hyperproperties. It would be interesting to discover additional 4-safety hyperproperties that would strengthen different aspects of information flow properties. One might also be compelled to study whether metaconditions of higher order would be useful. For example, a meta-metacondition could stipulate that if a system satisfies speculative noninterference against passive attackers, then this system should

satisfy speculative noninterference against active attackers, too. And this meta-metacondition seems to be an 8-safety hyperproperty.

Information Flow Properties and Fairness

Machine learning models are ubiquitous in the processing of users' data. To maintain control over user's data, one needs to understand how data is being processed by these models. Explainable machine learning is a way to achieve this understanding. Some consider *fairness* as an instance of explainable machine learning. But fairness can be interpreted as an information flow policy: changing the value of the sensitive feature should not change a particular statistical metric (e.g., positive predictive value) of the learned model. Establishing such connections will put the rich literature of information flow properties at the service of fairness.

Quantum Computation

Quantum computing offers a new and intriguing computational model. Within this model, information flows between entities in ways that are not possible within a traditional computer (e.g., through entanglement). Specifying which of these flows are allowed or forbidden seems challenging. Defining a threat model appropriate for quantum computing would be interesting, too. Here the mere leakage of some information from the system to external observers constitutes a measurement that changes the state of the entire system. With few exceptions, the interplay between quantum computation and information flow properties has been largely unexplored.

Acknowledgements

We would like to thank the reviewers for the valuable comments that improved the quality of this monograph. Martin Abadi, Fred B. Schneider, the PL groups at Cornell University and Harvard University gave helpful feedback and suggestions for the content of this monograph. We would also like to thank our editors Mike Casey and James Finlay for their support and encouragement throughout the writing process.

References

Abate, C., R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault. (2019). "Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation". In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE. 256–271. DOI: 10.1109/CSF.2019.00025.

Akella, R., H. Tang, and B. M. McMillin. (2010). "Analysis of information flow security in cyber-physical systems". *Int. J. Crit. Infrastructure Prot.* 3(3-4): 157–173. DOI: 10.1016/j.ijcip.2010.09.001.

Akl, S. G. and D. E. Denning. (1987). "Checking Classification Constraints for Consistency and Completeness". In: *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*. IEEE Computer Society. 196–201. DOI: 10.1109/SP.1987.10000.

Allen, P. G. (1991). "A Comparison of non-Interference and Non-Deducibility using CSP". In: *4th IEEE Computer Security Foundations Workshop - CSFW'91, Franconia, New Hampshire, USA, June 18-20, 1991, Proceedings*. IEEE Computer Society. 43–54. DOI: 10.1109/CSFW.1991.151568.

Alvim, M. S., M. E. Andrés, K. Chatzikokolakis, and C. Palamidessi. (2011). "On the Relation between Differential Privacy and Quantitative Information Flow". In: *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. Ed. by L. Aceto, M. Henzinger, and J. Sgall. Vol. 6756. *Lecture Notes in Computer Science*. Springer. 60–76. DOI: 10.1007/978-3-642-22012-8_4.

Alvim, M. S., M. E. Andrés, and C. Palamidessi. (2012a). "Quantitative information flow in interactive systems". *Journal of Computer Security*. 20(1): 3–50. DOI: 10.3233/JCS-2011-0433.

Alvim, M. S., K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith. (2014a). "Additive and Multiplicative Notions of Leakage, and Their Capacities". In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society. 308–322. DOI: 10.1109/CSF.2014.29.

Alvim, M. S., K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith. (2020). *The Science of Quantitative Information Flow*. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-96131-6.

Alvim, M. S., K. Chatzikokolakis, C. Palamidessi, and G. Smith. (2012b). "Measuring Information Leakage Using Generalized Gain Functions". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by S. Chong. IEEE Computer Society. 265–279. DOI: 10.1109/CSF.2012.26.

Alvim, M. S., P. Mardziel, and M. W. Hicks. (2017). "Quantifying Vulnerability of Secret Generation Using Hyper-Distributions". In: *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by M. Maffei and M. Ryan. Vol. 10204. *Lecture Notes in Computer Science*. Springer. 26–48. DOI: 10.1007/978-3-662-54455-6_2.

Alvim, M. S., A. Scedrov, and F. B. Schneider. (2014b). "When Not All Bits Are Equal: Worth-Based Information Flow". In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by M. Abadi and S. Kremer. Vol. 8414. *Lecture Notes in Computer Science*. Springer. 120–139. DOI: 10.1007/978-3-642-54792-8_7.

Amorim, A. A. de, N. Collins, A. DeHon, D. Demange, C. Hritcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. (2014). "A verified information-flow architecture". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by S. Jagannathan and P. Sewell. ACM. 165–178. DOI: 10.1145/2535838.2535839.

Arden, O., J. Liu, and A. C. Myers. (2015). "Flow-Limited Authorization". In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. Ed. by C. Fournet, M. W. Hicks, and L. Viganò. IEEE Computer Society. 569–583. DOI: 10.1109/CSF.2015.42.

Ashby, W. R. (1956). *An Introduction to Cybernetis*. Martino Fine Books (January 25, 2015).

Askarov, A. and S. Chong. (2012). "Learning is Change in Knowledge: Knowledge-Based Security for Dynamic Policies". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by S. Chong. IEEE Computer Society. 308–322. DOI: 10.1109/CSF.2012.31.

Askarov, A., S. Hunt, A. Sabelfeld, and D. Sands. (2008). "Termination-Insensitive Noninterference Leaks More Than Just a Bit". In: *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*. Ed. by S. Jajodia and J. López. Vol. 5283. *Lecture Notes in Computer Science*. Springer. 333–348. DOI: 10.1007/978-3-540-88313-5_22.

Askarov, A. and A. Myers. (2010). "A Semantic Framework for Declassification and Endorsement". In: *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by A. D. Gordon. Vol. 6012. *Lecture Notes in Computer Science*. Springer. 64–84. DOI: 10.1007/978-3-642-11957-6_5.

Askarov, A. and A. Sabelfeld. (2007a). "Gradual Release: Unifying Declassification, Encryption and Key Release Policies". In: *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*. IEEE Computer Society. 207–221. DOI: 10.1109/SP.2007.22.

Askarov, A. and A. Sabelfeld. (2007b). "Localized delimited release: combining the what and where dimensions of information release". In: *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007*. Ed. by M. W. Hicks. ACM. 53–60. DOI: 10.1145/1255329.1255339.

Askarov, A. and A. Sabelfeld. (2009). "Tight Enforcement of Information-Release Policies for Dynamic Languages". In: *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*. IEEE Computer Society. 43–59. DOI: 10.1109/CSF.2009.22.

Austin, T. H. and C. Flanagan. (2009). "Efficient purely-dynamic information flow analysis". In: *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*. Ed. by S. Chong and D. A. Naumann. ACM. 113–124. DOI: 10.1145/1554339.1554353.

Backes, M. (2005). "Quantifying Probabilistic Information Flow in Computational Reactive Systems". In: *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*. Ed. by S. D. C. di Vimercati, P. F. Syverson, and D. Gollmann. Vol. 3679. *Lecture Notes in Computer Science*. Springer. 336–354. DOI: 10.1007/11555827_20.

Backes, M. and B. Pfitzmann. (2002). "Computational Probabilistic Non-interference". In: *Computer Security - ESORICS 2002, 7th European Symposium on Research in Computer Security, Zurich, Switzerland, October 14-16, 2002, Proceedings*. Ed. by D. Gollmann, G. Karjoth, and M. Waidner. Vol. 2502. *Lecture Notes in Computer Science*. Springer. 1–23. DOI: 10.1007/3-540-45853-0_1.

Balliu, M. (2013). "A Logic for Information Flow Analysis of Distributed Programs". In: *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Ed. by H. R. Nielson and D. Gollmann. Vol. 8208. *Lecture Notes in Computer Science*. Springer. 84–99. DOI: 10.1007/978-3-642-41488-6_6.

Balliu, M., M. Dam, and G. L. Guernic. (2011). "Epistemic temporal logic for information flow security". In: *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011, San Jose, CA, USA, 5 June, 2011*. Ed. by A. Askarov and J. D. Guttman. ACM. 6. DOI: 10.1145/2166956.2166962.

Banerjee, A., D. A. Naumann, and S. Rosenberg. (2008). "Expressive Declassification Policies and Modular Static Enforcement". In: *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society. 339–353. DOI: 10.1109/SP.2008.20.

Becker, M. Y. (2010). "Information Flow in Credential Systems". In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society. 171–185. DOI: 10.1109/CSF.2010.19.

Bell, D. E. and L. J. LaPadula. (1973). "Secure Computer Systems: mathematical foundations and model". *Tech. rep.* No. M74-244. Bedford, MA: MITRE Corp.

Bergström, E. and R. Åhlfeldt. (2014). "Information Classification Issues". In: *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*. Ed. by K. Bernsmed and S. Fischer-Hübner. Vol. 8788. *Lecture Notes in Computer Science*. Springer. 27–41. DOI: 10.1007/978-3-319-11599-3_2.

Best, E. and P. Darondeau. (2012). "Deciding Selective Declassification of Petri Nets". In: *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*. Ed. by P. Degano and J. D. Guttman. Vol. 7215. *Lecture Notes in Computer Science*. Springer. 290–308. DOI: 10.1007/978-3-642-28641-4_16.

Biba, K. J. (1977). "Integrity Considerations for Secure Computer Systems". *Tech. rep.* No. ESD-TR-76-372. USAF Electronic Systems Division.

Birrell, E. and F. B. Schneider. (2017). "A Reactive Approach for Use-Based Privacy". *Tech. rep.* Cornell University.

Bishop, M. and L. Snyder. (1979). "The Transfer of Information and Authority in a Protection System". In: *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*. Ed. by M. D. Schroeder and A. K. Jones. ACM. 45–54. DOI: 10.1145/800215.806569.

Bohannon, A., B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. (2009). "Reactive noninterference". In: *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. Ed. by E. Al-Shaer, S. Jha, and A. D. Keromytis. ACM. 79–90. DOI: 10.1145/1653662.1653673.

Bohrer, B. and A. Platzer. (2018). "A Hybrid, Dynamic Logic for Hybrid-Dynamic Information Flow". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by A. Dawar and E. Grädel. ACM. 115–124. DOI: 10.1145/3209108.3209151.

Broberg, N., B. van Delft, and D. Sands. (2015). "The Anatomy and Facets of Dynamic Policies". In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. Ed. by C. Fournet, M. W. Hicks, and L. Viganò. IEEE Computer Society. 122–136. DOI: 10.1109/CSF.2015.16.

Broberg, N. and D. Sands. (2009). "Flow-sensitive semantics for dynamic information flow policies". In: *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*. Ed. by S. Chong and D. A. Naumann. ACM. 101–112. DOI: 10.1145/1554339.1554352.

Broberg, N. and D. Sands. (2010). "Paralocks: role-based information flow control and beyond". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by M. V. Hermenegildo and J. Palsberg. ACM. 431–444. DOI: 10.1145/1706299.1706349.

Bryans, J. W., M. Koutny, L. Mazaré, and P. Y. A. Ryan. (2008). "Opacity generalised to transition systems". *Int. J. Inf. Sec.* 7(6): 421–435. DOI: 10.1007/s10207-008-0058-x.

Bryce, C. (1997). "Security Engineering of Lattice-Based Policies". In: *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society. 195–208. DOI: 10.1109/CSFW.1997.596813.

Cecchetti, E., A. C. Myers, and O. Arden. (2017). "Nonmalleable Information Flow Control". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM. 1875–1891. DOI: 10.1145/3133956.3134054.

Cecchetti, E., S. Yao, H. Ni, and A. C. Myers. (2021). "Compositional security for reentrant applications". In: *IEEE Symp. on Security and Privacy*.

Cheang, K., C. Rasmussen, S. A. Seshia, and P. Subramanyan. (2019). "A Formal Approach to Secure Speculation". In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE. 288–303. DOI: 10.1109/CSF.2019.00027.

Chen, H. and P. Malacaria. (2007). "Quantitative analysis of leakage for multi-threaded programs". In: *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007*. Ed. by M. W. Hicks. ACM. 31–40. DOI: 10.1145/1255329.1255335.

Chen, H. and S. Chong. (2004). "Owned Policies for Information Security". In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society. 126–138. DOI: 10.1109/CSFW.2004.15.

Cheng, W., D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. (2012). "Abstractions for Usable Information Flow Control in Aeolus". In: URL: http://dl.acm.org/citation.cfm?id=2342833.

Chong, S. (2010). "Required Information Release". In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society. 215–227. DOI: 10.1109/CSF.2010.22.

Chong, S. and A. C. Myers. (2006). "Decentralized Robustness". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society. 242–256. DOI: 10.1109/CSFW.2006.11.

Chong, S. and A. C. Myers. (2008). "End-to-End Enforcement of Erasure and Declassification". In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*. IEEE Computer Society. 98–111. DOI: 10.1109/CSF.2008.12.

Chong, S., K. Vikram, and A. C. Myers. (2007). "SIF: Enforcing Confidentiality and Integrity in Web Applications". In: *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. Ed. by N. Provos. USENIX Association. URL: https://www.usenix.org/conference/16th-usenix-security-symposium/sif-enforcing-confidentiality-and-integrity-web.

Chudnov, A., G. Kuan, and D. A. Naumann. (2014). "Information Flow Monitoring as Abstract Interpretation for Relational Logic". In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society. 48–62. DOI: 10.1109/CSF.2014.12.

Chudnov, A. and D. A. Naumann. (2018). "Assuming You Know: Epistemic Semantics of Relational Annotations for Expressive Flow Policies". In: *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society. 189–203. DOI: 10.1109/CSF.2018.00021.

Clark, D. and S. Hunt. (2008). "Non-Interference for Deterministic Interactive Programs". In: *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008, Malaga, Spain, October 9-10, 2008, Revised Selected Papers*. Ed. by P. Degano, J. D. Guttman, and F. Martinelli. Vol. 5491. *Lecture Notes in Computer Science*. Springer. 50–66. DOI: 10.1007/978-3-642-01465-9_4.

Clarkson, M. R., B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. (2014). "Temporal Logics for Hyperproperties". In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by M. Abadi and S. Kremer. Vol. 8414. *Lecture Notes in Computer Science*. Springer. 265–284. DOI: 10.1007/978-3-642-54792-8_15.

Clarkson, M. R., A. C. Myers, and F. B. Schneider. (2005). "Belief in Information Flow". In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society. 31–45. DOI: 10.1109/CSFW.2005.10.

Clarkson, M. R. and F. B. Schneider. (2010a). "Hyperproperties". *Journal of Computer Security*. 18(6): 1157–1210. DOI: 10.3233/JCS-2009-0393.

Clarkson, M. R. and F. B. Schneider. (2010b). "Quantification of Integrity". In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society. 28–43. DOI: 10.1109/CSF.2010.10.

Cohen, E. S. (1976). "Strong dependency : a formalism for describing information transmission in computational systems". *Tech. rep.* Carnegie Mellon University.

Cohen, E. S. (1977). "Information Transmission in Computational Systems". *ACM SIGOPS Operating Systems Review*. 11(5): 133–139.

Costanzo, D., Z. Shao, and R. Gu. (2016). "End-to-end verification of information-flow security for C and assembly programs". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by C. Krintz and E. Berger. ACM. 648–664. DOI: 10.1145/2908080.2908100.

Cruz, R., T. Rezk, B. P. Serpette, and É. Tanter. (2017). "Type Abstraction for Relaxed Noninterference". In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 7:1–7:27. DOI: 10.4230/LIPIcs.ECOOP.2017.7.

Dawson, S., S. D. C. di Vimercati, and P. Samarati. (1999). "Specification and Enforcement of Classification and Inference Constraints". In: *1999 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 9-12, 1999*. IEEE Computer Society. 181–195. DOI: 10.1109/SECPRI.1999.766913.

Denning, D. E. (1975). "Secure Information Flow in Computer Systems". *PhD thesis*. W. Lafayette, Indiana, USA: Purdue University.

Denning, D. E. (1976). "A Lattice Model of Secure Information Flow". *Communications of the ACM*. 19(5): 236–243.

Dimitrova, R., B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. (2012). "Model Checking Information Flow in Reactive Systems". In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*. Ed. by V. Kuncak and A. Rybalchenko. Vol. 7148. *Lecture Notes in Computer Science*. Springer. 169–185. DOI: 10.1007/978-3-642-27940-9_12.

Dimoulas, C., S. Moore, A. Askarov, and S. Chong. (2014). "Declarative Policies for Capability Control". In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society. 3–17. DOI: 10.1109/CSF.2014.9.

Do, Q. H., R. Bubel, and R. Hähnle. (2016). "Automatic detection and demonstrator generation for information flow leaks in object-oriented programs". *Computers & Security*. DOI: http://dx.doi.org/10.1016/j.cose.2016.12.002.

Dwork, C. (2006). "Differential Privacy". In: *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*. Ed. by M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener. Vol. 4052. *Lecture Notes in Computer Science*. Springer. 1–12. DOI: 10.1007/11787006_1.

Efstathopoulos, P., M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. (2005). "Labels and Event Processes in the Asbestos Operating System". In: Brighton, UK. URL: http://dl.acm.org/citation.cfm?id=1095813.

Engelhardt, K., R. van der Meyden, and C. Zhang. (2012). "Intransitive noninterference in nondeterministic systems". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. Ed. by T. Yu, G. Danezis, and V. D. Gligor. ACM. 869–880. DOI: 10.1145/2382196.2382288.

Ferraiuolo, A., M. Zhao, A. C. Myers, and G. E. Suh. (2018). "Hyperflow: A processor architecture for nonmalleable, timing-safe information-flow security". In: *25th ACM Conf. on Computer and Communications Security (CCS)*. URL: http://www.cs.cornell.edu/andru/papers/hyperflow.

Focardi, R. and R. Gorrieri. (1995). "A Taxonomy of Security Properties for Process Algebras". *Journal of Computer Security*. 3(1): 5–34. DOI: 10.3233/JCS-1994/1995-3103.

Focardi, R. and S. Rossi. (2002). "Information Flow Security in Dynamic Contexts". In: *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society. 307–319. DOI: 10.1109/CSFW.2002.1021825.

Foley, S. N. (1989). "A Model for Secure Information Flow". In: *Proceedings of the 1989 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 1-3, 1989*. IEEE Computer Society. 248–258. DOI: 10.1109/SECPRI.1989.36299.

Foley, S. N. (1991). "A Taxonomy for Information Flow Policies and Models". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society. 98–108.

Fournet, C. and T. Rezk. (2008). "Cryptographically sound implementations for typed information-flow security". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by G. C. Necula and P. Wadler. ACM. 323–335. DOI: 10.1145/1328438.1328478.

Glasgow, J. I. and G. H. MacEwen. (1989). "Obligation as the Basis of Integrity Specification". In: *Second IEEE Computer Security Foundations Workshop - CSFW'89, Franconia, New Hampshire, USA, June 11-14, 1989, Proceedings*. IEEE Computer Society. 64–70. DOI: 10.1109/CSFW.1989.40588.

Goguen, J. A. and J. Meseguer. (1982). "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society. 11–20. DOI: 10.1109/SP.1982.10014.

Greiner, S. and D. Grahl. (2016). "Non-interference with What-Declassification in Component-Based Systems". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society. 253–267. DOI: 10.1109/CSF.2016.25.

Guarnieri, M., B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. (2020). "Spectector: Principled Detection of Speculative Information Flows". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE. 1–19. DOI: 10.1109/SP40000.2020.00011.

Guri, M., Y. A. Solewicz, A. Daidakulov, and Y. Elovici. (2017). "Acoustic Data Exfiltration from Speakerless Air-Gapped Computers via Covert Hard-Drive Noise ('DiskFiltration')". In: *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*. Ed. by S. N. Foley, D. Gollmann, and E. Snekkenes. Vol. 10493. *Lecture Notes in Computer Science*. Springer. 98–115. DOI: 10.1007/978-3-319-66399-9_6.

Guttman, J. D. and P. D. Rowe. (2015). "A Cut Principle for Information Flow". In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. Ed. by C. Fournet, M. W. Hicks, and L. Viganò. IEEE Computer Society. 107–121. DOI: 10.1109/CSF.2015.15.

Halpern, J. Y. and K. R. O'Neill. (2002). "Secrecy in Multiagent Systems". In: *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society. 32. DOI: 10.1109/CSFW.2002.1021805.

Halpern, J. Y. and K. R. O'Neill. (2008). "Secrecy in Multiagent Systems". *ACM Trans. Inf. Syst. Secur.* 12(1): 5:1–5:47.

Hamadou, S., V. Sassone, and C. Palamidessi. (2010). "Reconciling Belief and Vulnerability in Information Flow". In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society. 79–92. DOI: 10.1109/SP.2010.13.

Hartman, B. (1988). "A General Approach to Tranquility in Information Flow Analysis". In: *First IEEE Computer Security Foundations Workshop - CSFW'88, Franconia, New Hampshire, USA, June 12-15, 1988, Proceedings*. MITRE Corporation Press. 166–181.

Heintze, N. and J. G. Riecke. (1998). "The SLam Calculus: Programming with Secrecy and Integrity". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by D. B. MacQueen and L. Cardelli. ACM. 365–377. DOI: 10.1145/268946.268976.

Hicks, B., D. King, P. McDaniel, and M. Hicks. (2006). "Trusted declassification: : high-level policy for a security-typed language". In: *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, PLAS 2006, Ottawa, Ontario, Canada, June 10, 2006*. Ed. by V. C. Sreedhar and S. Zdancewic. ACM. 65–74. DOI: 10.1145/1134744.1134757.

Hughes, D. J. D. and V. Shmatikov. (2004). "Information Hiding, Anonymity and Privacy: a Modular Approach". *Journal of Computer Security*. 12(1): 3–36. URL: http://content.iospress.com/articles/journal-of-computer-security/jcs197.

Hunt, S. and D. Sands. (2008). "Just Forget It - The Semantics and Enforcement of Information Erasure". In: *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by S. Drossopoulou. Vol. 4960. *Lecture Notes in Computer Science*. Springer. 239–253. DOI: 10.1007/978-3-540-78739-6_19.

Jaume, M. (2012). "Semantic Comparison of Security Policies: From Access Control Policies to Flow Properties". In: *2012 IEEE Symposium on Security and Privacy Workshops, San Francisco, CA, USA, May 24-25, 2012*. 60–67. DOI: 10.1109/SPW.2012.33.

Kaneko, Y. and N. Kobayashi. (2008). "Linear Declassification". In: *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by S. Drossopoulou. Vol. 4960. *Lecture Notes in Computer Science*. Springer. 224–238. DOI: 10.1007/978-3-540-78739-6_18.

Kashyap, V., B. Wiedermann, and B. Hardekopf. (2011). "Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach". In: *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society. 413–428. DOI: 10.1109/SP.2011.19.

Kocher, P. C. (1996). "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Ed. by N. Koblitz. Vol. 1109. *Lecture Notes in Computer Science*. Springer. 104–113. DOI: 10.1007/3-540-68697-5_9.

Köpf, B. and D. A. Basin. (2007). "An information-theoretic model for adaptive side-channel attacks". In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. Ed. by P. Ning, S. D. C. di Vimercati, and P. F. Syverson. ACM. 286–296. DOI: 10.1145/1315245.1315282.

Kozyri, E. and F. B. Schneider. (2020). "RIF: Reactive information flow labels". *J. Comput. Secur.* 28(2): 191–228. DOI: 10.3233/JCS-191316.

Kozyri, E., F. B. Schneider, A. Bedford, J. Desharnais, and N. Tawbi. (2019). "Beyond Labels: Permissiveness for Dynamic Information Flow Enforcement". In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE. 351–366. DOI: 10.1109/CSF.2019.00031.

Krohn, M. N. and E. Tromer. (2009). "Noninterference for a Practical DIFC-Based Operating System". In: *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society. 61–76. DOI: 10.1109/SP.2009.23.

Krohn, M. N., A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. (2007). "Information flow control for standard OS abstractions". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. Ed. by T. C. Bressoud and M. F. Kaashoek. ACM. 321–334. DOI: 10.1145/1294261.1294293.

Lampson, B. W. (1973). "A note on the confinement problem". *Communications of the ACM*. 16(10): 613–615.

Laud, P. (2001). "Semantics and Program Analysis of Computationally Secure Information Flow". In: *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by D. Sands. Vol. 2028. *Lecture Notes in Computer Science*. Springer. 77–91. DOI: 10.1007/3-540-45309-1_6.

Lewis, D. (1973). "Causation". *Journal of Philosophy*. 70(17): 556–567. DOI: 10.2307/2025310.

Li, P. and D. Zhang. (2021). "Towards a General-Purpose Dynamic Information Flow Policy". *CoRR*. abs/2109.08096. arXiv: 2109.08096. URL: https://arxiv.org/abs/2109.08096.

Li, P., Y. Mao, and S. Zdancewic. (2003). "Information integrity policies". In: *Proceedings of the Workshop on Formal Aspects in Security and Trust*.

Li, P. and S. Zdancewic. (2005a). "Downgrading policies and relaxed noninterference". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by J. Palsberg and M. Abadi. ACM. 158–170. DOI: 10.1145/1040305.1040319.

Li, P. and S. Zdancewic. (2005b). "Unifying Confidentiality and Integrity in Downgrading Policies". In: *Proceedings of the Foundations of Computer Security Workshop*.

Liu, J., M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. (2009). "Fabric: a platform for secure distributed computation and storage". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by J. N. Matthews and T. E. Anderson. ACM. 321–334. DOI: 10.1145/1629575.1629606.

Lu, Y. and C. Zhang. (2020). "Nontransitive Security Types for Coarse-grained Information Flow Control". In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE. 199–213. DOI: 10.1109/CSF49147.2020.00022.

Lux, A. and H. Mantel. (2009). "Declassification with Explicit Reference Points". In: *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*. Ed. by M. Backes and P. Ning. Vol. 5789. *Lecture Notes in Computer Science*. Springer. 69–85. DOI: 10.1007/978-3-642-04444-1_5.

Magazinius, J., A. Askarov, and A. Sabelfeld. (2010). "A lattice-based approach to mashup security". In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*. Ed. by D. Feng, D. A. Basin, and P. Liu. ACM. 15–23. DOI: 10.1145/1755688.1755691.

Malacaria, P. (2007). "Assessing security threats of looping constructs". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. Ed. by M. Hofmann and M. Felleisen. ACM. 225–235. DOI: 10.1145/1190216.1190251.

Malacaria, P. and H. Chen. (2008). "Lagrange multipliers and maximum information leakage in different observational models". In: *Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security, PLAS 2008, Tucson, AZ, USA, June 8, 2008*. Ed. by Ú. Erlingsson and M. Pistoia. ACM. 135–146. DOI: 10.1145/1375696.1375713.

Mantel, H. (2002). "On the Composition of Secure Systems". In: *2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 12-15, 2002*. IEEE Computer Society. 88–101. DOI: 10.1109/SECPRI.2002.1004364.

Mantel, H., M. Perner, and J. Sauer. (2014). "Noninterference under Weak Memory Models". In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society. 80–94. DOI: 10.1109/CSF.2014.14.

Mantel, H., D. Sands, and H. Sudbrock. (2011). "Assumptions and Guarantees for Compositional Noninterference". In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society. 218–232. DOI: 10.1109/CSF.2011.22.

Mantel, H. and H. Sudbrock. (2010). "Flexible Scheduler-Independent Security". In: *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*. Ed. by D. Gritzalis, B. Preneel, and M. Theoharidou. Vol. 6345. *Lecture Notes in Computer Science*. Springer. 116–133. DOI: 10.1007/978-3-642-15497-3_8.

Mardziel, P., M. S. Alvim, M. W. Hicks, and M. R. Clarkson. (2014). "Quantifying Information Flow for Dynamic Secrets". In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society. 540–555. DOI: 10.1109/SP.2014.41.

Masti, R. J., D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. (2015). "Thermal Covert Channels on Multi-core Platforms". In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Ed. by J. Jung and T. Holz. USENIX Association. 865–880. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti.

McCullough, D. (1988). "Noninterference and the composability of security properties". In: *Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 18-21, 1988*. IEEE Computer Society. 177–186. DOI: 10.1109/SECPRI.1988.8110.

McLean, J. (1990a). "Security Models and Information Flow". In: *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*. IEEE Computer Society. 180–189. DOI: 10.1109/RISP.1990.63849.

McLean, J. (1990b). "The Specification and Modeling of Computer Security". *Computer*. 23(1): 9–16. DOI: 10.1109/2.48795.

McLean, J. (1994). "A general theory of composition for trace sets closed under selective interleaving functions". In: *1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 16-18, 1994*. IEEE Computer Society. 79–93. DOI: 10.1109/RISP.1994.296590.

Meadows, C. A. (1990). "Extending the Brewer-Nash Model to a Multilevel Context". In: *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*. IEEE Computer Society. 95–103. DOI: 10.1109/RISP.1990.63842.

Mestel, D. (2019). "Quantifying Information Flow in Interactive Systems". In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE. 414–427. DOI: 10.1109/CSF.2019.00035.

Micinski, K. K., J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson. (2015). "Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution". In: *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*. Ed. by G. Pernul, P. Y. A. Ryan, and E. R. Weippl. Vol. 9327. *Lecture Notes in Computer Science*. Springer. 520–538. DOI: 10.1007/978-3-319-24177-7_26.

Millen, J. K. (1987). "Covert Channel Capacity". In: *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*. IEEE Computer Society. 60–66. DOI: 10.1109/SP.1987.10013.

Montagu, B., B. C. Pierce, and R. Pollack. (2013). "A Theory of Information-Flow Labels". In: *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*. IEEE Computer Society. 3–17. DOI: 10.1109/CSF.2013.8.

Moore, S. and S. Chong. (2011). "Static Analysis for Efficient Hybrid Information-Flow Control". In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society. 146–160. DOI: 10.1109/CSF.2011.17.

Mundie, C. (2014). "Privacy Pragmatism: Focus on Data Use, Not Data Collection". *Foreign Affairs*. 93(2): 28–38. URL: http://www.jstor.org/stable/24483581.

Myers, A. C. and B. Liskov. (1997). "A Decentralized Model for Information Flow Control". In: *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*. Ed. by M. Banâtre, H. M. Levy, and W. M. Waite. ACM. 129–142. DOI: 10.1145/268998.266669.

Myers, A. C., A. Sabelfeld, and S. Zdancewic. (2006). "Enforcing Robust Declassification and Qualified Robustness". *Journal of Computer Security*. 14(2): 157–196. URL: http://content.iospress.com/articles/journal-of-computer-security/jcs258.

Nanevski, A., A. Banerjee, and D. Garg. (2013). "Dependent Type Theory for Verification of Information Flow and Access Control Policies". *ACM Trans. Program. Lang. Syst.* 35(2): 6:1–6:41. DOI: 10.1145/2491522.2491523.

Nissenbaum, H. (2010). *Privacy in Context - Technology, Policy, and the Integrity of Social Life*. Stanford University Press. URL: http://www.sup.org/book.cgi?id=8862.

O'Halloran, C. (1990). "A Calculus of Information Flow". In: *ESORICS 90 - First European Symposium on Research in Computer Security, October 24-26, 1990, Toulouse, France*. AFCET. 147–159.

O'Neill, K. R., M. R. Clarkson, and S. Chong. (2006). "Information-Flow Security for Interactive Programs". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society. 190–201. DOI: 10.1109/CSFW.2006.16.

Patrignani, M., A. Ahmed, and D. Clarke. (2019). "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work". *ACM Comput. Surv.* 51(6). DOI: 10.1145/3280984.

Pedersen, M. L., M. H. Sørensen, D. Lux, U. Nyman, and R. R. Hansen. (2015). "The Timed Decentralised Label Model". In: *Secure IT Systems, 20th Nordic Conference, NordSec 2015, Stockholm, Sweden, October 19-21, 2015, Proceedings*. Ed. by S. Buchegger and M. Dam. Vol. 9417. *Lecture Notes in Computer Science*. Springer. 27–43. DOI: 10.1007/978-3-319-26502-5_3.

Rafnsson, W. and A. Sabelfeld. (2014). "Compositional Information-Flow Security for Interactive Systems". In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society. 277–292. DOI: 10.1109/CSF.2014.27.

Rakotonirina, I. and B. Köpf. (2019). "On Aggregation of Information in Timing Attacks". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. 387–400. DOI: 10.1109/EuroSP.2019.00036.

Rocha, B. P. S., S. Bandhakavi, J. den Hartog, W. H. Winsborough, and S. Etalle. (2010). "Towards Static Flow-Based Declassification for Legacy and Untrusted Programs". In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society. 93–108. DOI: 10.1109/SP.2010.14.

Roscoe, A. W. (1995). "CSP and determinism in security modelling". In: *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*. IEEE Computer Society. 114–127. DOI: 10.1109/SECPRI.1995.398927.

Roy, I., D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. (2009). "Laminar: practical fine-grained decentralized information flow control". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. by M. Hind and A. Diwan. ACM. 63–74. DOI: 10.1145/1542476.1542484.

Rushby, J. (1992). "Noninterference, transitivity and channel-control security policies". *Tech. rep.*

Rushby, J. M. (1981). "Design and Verification of Secure Systems". In: *Proceedings of the Eighth Symposium on Operating System Principles, SOSP 1981, Asilomar Conference Grounds, Pacific Grove, California, USA, December 14-16, 1981*. Ed. by J. Howard and D. P. Reed. ACM. 12–21. DOI: 10.1145/800216.806586.

Russo, A. and A. Sabelfeld. (2006). "Securing Interaction between Threads and the Scheduler". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society. 177–189. DOI: 10.1109/CSFW.2006.29.

Sabelfeld, A. and A. C. Myers. (2003a). "A Model for Delimited Information Release". In: *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*. Ed. by K. Futatsugi, F. Mizoguchi, and N. Yonezaki. Vol. 3233. *Lecture Notes in Computer Science*. Springer. 174–191. DOI: 10.1007/978-3-540-37621-7_9.

Sabelfeld, A. and A. C. Myers. (2003b). "Language-Based Information-Flow Security". *IEEE Journal on Selected Areas in Communications*. 21(1): 5–19.

Sabelfeld, A. and D. Sands. (2000). "Probabilistic Noninterference for Multi-Threaded Programs". In: *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*. IEEE Computer Society. 200–214. DOI: 10.1109/CSFW.2000.856937.

Sabelfeld, A. and D. Sands. (2005). "Dimensions and Principles of Declassification". In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society. 255–269. DOI: 10.1109/CSFW.2005.15.

Sabelfeld, A. and D. Sands. (2009). "Declassification: Dimensions and principles". *Journal of Computer Security*. 17(5): 517–548. DOI: 10.3233/JCS-2009-0352.

Schoepe, D. and A. Sabelfeld. (2015). "Understanding and Enforcing Opacity". In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. Ed. by C. Fournet, M. W. Hicks, and L. Viganò. IEEE Computer Society. 539–553. DOI: 10.1109/CSF.2015.41.

Schultz, D. A. and B. Liskov. (2013). "IFDB: decentralized information flow control for databases". In: *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*. Ed. by Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek. ACM. 43–56. DOI: 10.1145/2465351.2465357.

Shannon, C. E. (1948). "A mathematical theory of communication". *The Bell System Technical Journal*. 27(3): 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

Smith, G. (2006). "Improved typings for probabilistic noninterference in a multi-threaded language". *Journal of Computer Security*. 14(6): 591–623. URL: http://content.iospress.com/articles/journal-of-computer-security/jcs273.

Smith, G. and D. M. Volpano. (1998). "Secure Information Flow in a Multi-Threaded Imperative Language". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by D. B. MacQueen and L. Cardelli. ACM. 355–364. DOI: 10.1145/268946.268975.

Stefan, D., A. Russo, D. Mazières, and J. C. Mitchell. (2011a). "Disjunction Category Labels". In: *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers*. Ed. by P. Laud. Vol. 7161. *Lecture Notes in Computer Science*. Springer. 223–239. DOI: 10.1007/978-3-642-29615-4_16.

Stefan, D., A. Russo, J. C. Mitchell, and D. Mazières. (2011b). "Flexible Dynamic Information Flow Control in Haskell". In: *Haskell Symposium*. ACM SIGPLAN. URL: http://doi.acm.org/10.1145/2096148.2034688.

Stoughton, A. (1981). "Access Flow: A Protection Model which Integrates Access Control and Information Flow". In: *1981 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 27-29, 1981*. IEEE Computer Society. 9–18. DOI: 10.1109/SP.1981.10004.

Sutherland, D. (1986). "A model of information". In: *Proceedings of the 9th National Security Conference*. 175–183.

Swamy, N., M. Hicks, S. Tse, and S. Zdancewic. (2006). "Managing Policy Updates in Security-Typed Languages". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society. 202–216. DOI: 10.1109/CSFW.2006.17.

Tedesco, F. D., D. Sands, and A. Russo. (2016). "Fault-Resilient Noninterference". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society. 401–416. DOI: 10.1109/CSF.2016.35.

Tschantz, M. C., S. Sen, and A. Datta. (2020). "SoK: Differential Privacy as a Causal Property". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE. 354–371. DOI: 10.1109/SP40000.2020.00012.

van der Meyden, R. (2007). "What, Indeed, Is Intransitive Noninterference?" In: *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*. Ed. by J. Biskup and J. Lopez. Vol. 4734. *Lecture Notes in Computer Science*. Springer. 235–250. DOI: 10.1007/978-3-540-74835-9_16.

Vanhoef, M., W. D. Groef, D. Devriese, F. Piessens, and T. Rezk. (2014). "Stateful Declassification Policies for Event-Driven Programs". In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society. 293–307. DOI: 10.1109/CSF.2014.28.

Vaughan, J. A. and T. D. Millstein. (2012). "Secure Information Flow for Concurrent Programs under Total Store Order". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by S. Chong. IEEE Computer Society. 19–29. DOI: 10.1109/CSF.2012.20.

Volpano, D. M., C. E. Irvine, and G. Smith. (1996). "A Sound Type System for Secure Flow Analysis". *Journal of Computer Security*. 4(2/3): 167–188. DOI: 10.3233/JCS-1996-42-304.

Volpano, D. M. and G. Smith. (1997). "Eliminating Covert Flows with Minimum Typings". In: *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society. 156–169. DOI: 10.1109/CSFW.1997.596807.

Volpano, D. M. and G. Smith. (1999). "Probabilistic Noninterference in a Concurrent Language". *Journal of Computer Security*. 7(1). URL: http://content.iospress.com/articles/journal-of-computer-security/jcs129.

Wittbold, J. T. and D. M. Johnson. (1990). "Information Flow in Nondeterministic Systems". In: *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*. IEEE Computer Society. 144–161. DOI: 10.1109/RISP.1990.63846.

Woodward, J. P. L. (1987). "Exploiting the Dual Nature of Sensitivity Labels". In: *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*. IEEE Computer Society. 23–31. DOI: 10.1109/SP.1987.10016.

Ying, M., Y. Feng, and N. Yu. (2013). "Quantum Information-Flow Security: Noninterference and Access Control". In: *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*. IEEE Computer Society. 130–144. DOI: 10.1109/CSF.2013.16.

Zakinthinos, A. and E. S. Lee. (1995). "The Composability of Non-Interference". *Journal of Computer Security*. 3(4): 269–282. DOI: 10.3233/JCS-1994/1995-3404.

Zakinthinos, A. and E. S. Lee. (1996). "How and why feedback composition fails [secure systems]". In: *Ninth IEEE Computer Security Foundations Workshop, March 10 - 12, 1996, Dromquinna Manor, Kenmare, County Kerry, Ireland*. IEEE Computer Society. 95–101. DOI: 10.1109/CSFW.1996.503694.

Zakinthinos, A. and E. S. Lee. (1997). "A General Theory of Security Properties". In: *1997 IEEE Symposium on Security and Privacy, May 4-7, 1997, Oakland, CA, USA*. IEEE Computer Society. 94–102. DOI: 10.1109/SECPRI.1997.601322.

Zdancewic, S. and A. C. Myers. (2001). "Robust Declassification". In: *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society. 15. DOI: 10.1109/CSFW.2001.930133.

Zdancewic, S. and A. C. Myers. (2003). "Observational Determinism for Concurrent Program Security". In: *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*. IEEE Computer Society. 29. DOI: 10.1109/CSFW.2003.1212703.

Zdancewic, S., L. Zheng, N. Nystrom, and A. C. Myers. (2001). "Untrusted Hosts and Confidentiality: Secure Program Partitioning". In: *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*. Ed. by K. Marzullo and M. Satyanarayanan. ACM. 1–14. DOI: 10.1145/502034.502036.

Zeldovich, N., S. Boyd-Wickizer, E. Kohler, and D. Mazières. (2006). "Making Information Flow Explicit in HiStar". In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. Ed. by B. N. Bershad and J. C. Mogul. USENIX Association. 263–278. URL: http://www.usenix.org/events/osdi06/tech/zeldovich.html.

Zeldovich, N., S. Boyd-Wickizer, and D. Mazières. (2008). "Securing Distributed Systems with Information Flow Control". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA: USENIX Association. 293–308.

Zheng, L. and A. C. Myers. (2005). "End-to-End Availability Policies and Noninterference". In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society. 272–286. DOI: 10.1109/CSFW.2005.16.

Zheng, L. and A. C. Myers. (2007). "Dynamic security labels and static information flow control". *Int. J. Inf. Sec.* 6(2-3): 67–84. DOI: 10.1007/s10207-007-0019-9.

Zheng, L. and A. C. Myers. (2014). "A Language-Based Approach to Secure Quorum Replication". In: *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014*. Ed. by A. Russo and O. Tripp. ACM. 27. DOI: 10.1145/2637113.2637117.