A Perspective on Information-Flow Control

Daniel HEDIN [a] and Andrei SABELFELD [a]
[a] *Chalmers University of Technology, Gothenburg, Sweden*

Abstract. Information-flow control tracks how information propagates through the program during execution to make sure that the program handles the information securely. Secure information flow is comprised of two related aspects: information confidentiality and information integrity — intuitively pertaining to the reading and writing of the information. The prevailing basic semantic notion of secure information flow is noninterference, demanding independence of public (or, in the case of integrity, trusted) output from secret (or, in the case of integrity, untrusted) input. This document gives an account of the state-of-the-art in confidentiality and integrity policies and their enforcement with a systematic formalization of four dominant formulations of noninterference: termination-insensitive, termination-sensitive, progress-insensitive, and progress-sensitive, cast in the setting of two minimal while languages.

1. Information-flow control

The control of how information is propagated by computing systems is vital for information security. Historically, *access control* has been the main means of preventing information from being disseminated. As the name indicates, access control verifies that the program's access rights at the point of access, and either grants or denies the program access. Once the program has been given access to information no further effort is made to make sure that the program handles the accessed information correctly. However, access control is inadequate in many situations, since it forces an all-or-nothing choice of either fully trusting the program not to leak/compromise information or not allowing access to this information altogether.

Information-flow control tracks how information propagates through the program during execution to make sure that the program handles the information securely. The research on secure information flow goes back to the early 70's [35,39], primarily in the context of military systems. Secure information flow is comprised of two related aspects: information *confidentiality* and information *integrity* — intuitively pertaining to the reading and writing of the information. The prevailing basic semantic notion of secure information flow is *noninterference* [46], demanding independence of public (or, in the case of integrity, trusted) output from secret (or, in the case of integrity, untrusted) input. As the field has matured, numerous variations of noninterference [98], as well as other semantic characterizations have been explored [103].

Recently, information integrity has received attention [55,57,19,4]. Integrity has frequently been seen as the dual of confidentiality [18], though it can be argued that this description might ignore other important facets [19].

One important aspect of integrity lies in its interaction with *declassification* — intentional lowering of security classification of information — in order to prevent the attacker from controlling what information is declassified [77,78].

Below we give an account of the state-of-the-art in confidentiality and integrity policies and enforcement, with a detailed exposition of various formulations of noninterference.

1.1. Attacker model

Information-flow security aims at protecting confidentiality and integrity of information. Traditionally, the attacker model is program-centric; the attacker is assumed to have access to the program source code and to public observable behavior, e.g., public outputs. In addition, it is assumed that the attacker is in control of the public input of the programs.

With the rise of the web as a more general application platform the old program-centric attacker model is no longer adequate. A web application is often a *mashup* (programmableweb.com) of information and services from different distributed sources. In the web setting, a notion of *web attacker* [15] is of interest. The model is built on the assumption of an honest user who runs a trusted browser on a trusted machine and that the attacker is an owner of malicious web sites that the user might be accessing. This implies that the web attacker is unable to mount man-in-the-middle attacks. This is in contrast to the the classical Dolev-Yao attacker where the attacker is able to overhear, intercept and modify messages on the network. Instead, the network capabilities of the web attacker are restricted to the standard network protocols of communication with servers in the domain of attacker's control.

1.2. Policy languages for confidentiality and integrity

Both confidentiality and integrity policies can be staged into two parts. The *policy language* describing the possible classifications of information and how the different classifications relate, i.e., how information from one classification may flow with respect to the other and the *semantic characterization* describing the meaning of the policy language in terms of the semantics of the programming language. The former is predominantly described using a lattice model [38,52] for information classification. Historically, the latter has typically been a variant of noninterference.

Deviating from policy language based on a lattice of security levels is the work by Myers and Liskov on the *Decentralized Label Model* (DLM) [76]. The DLM shares the lattice structure of classifications, but differs in the intentional interpretation of security levels. The security labels of the DLM track information *ownership* and *read rights* — a label $\{A : B\}$ expresses that A owns the data and that B is given read rights to the data.

More fundamentally different is the policy language of FlowLocks [22,23,24]. The core idea is to associate data with a set of clauses of the form $\Sigma \rightarrow A$. The set Σ states the circumstances under which Amay view the data.

1.3. Outline and sources

This document is laid out as follows. Section 2 investigates noninterference for confidentiality. In particular, we detail and compare four flavors of noninterference in terms of two small while languages, and discuss their relative merits. The section ends with a

discussion on declassification, and the different dimensions of declassification: *what* is declassified, *who* is able to declassify, *where* the declassification occurs and *when* the declassification takes place. Section 3 presents different facets of integrity: integrity *as a dual to confidentiality* and integrity *as program correctness*. The section ends with a discussion on endorsement (the dual to declassification, when integrity is taken to be the dual of confidentiality), and the corresponding interpretation of the four dimensions. Finally, Section 4 discusses different methods for enforcing secure information flow. In particular, dynamic, static and hybrid analyses are contrasted.

Given the tutorial nature of this document, we borrow some material and exposition from a few sources [103,100,64,19,117]. Yet the systematic formalization of noninterference policies is presented here for the first time. Because this document is more a tutorial than a survey, we discuss a selection of the area's highlights rather than an exhaustive account of all work on the topic.

2. Confidentiality

An important aspect of information-flow security is the preservation of confidentiality of information. In this section we will explore security policies for confidentiality with focus on noninterference. There are various flavors of noninterference offering different levels of assurance depending on the power of the attacker and the features of the underlying programming language. The different variants occur in the literature under a variety of different, sometimes overlapping names. We describe the landscape of noninterference definitions according to the following four dominant axes: termination-insensitive vs. termination-sensitive noninterference and progress-insensitive vs. progress-sensitive noninterference.

2.1. Noninterference

A program may during execution — advertently or inadvertently — leak information. To ensure secure information flow we need to consider *end-to-end* security, which involves not only preventing unauthorized access to information, but also tracking how information is flowing through the program during execution.

2.1.1. Sources of information flow

There are two principal sources of information flow in programs. *Explicit flows* correspond to the direct copying of secrets — when the value of one variable is copied into another. The following code fragment copies the value of a secret (or *high*) variable h into a public (or *low*) variable l:

```
l = h
```

Implicit flows [39] may arise when the control flow is affected by secret values; any differences in side effects under secret control encode information about the control. Consider, for instance, the following small example where the value of h is copied indirectly into l:

```
l = false; if h then l = true else skip
```

As foreshadowed earlier, the dominant semantic characterization of secure information flow is in terms of *noninterference*. The term noninterference was coined by Goguen and Meseguer [46] but goes back to the notion of *strong dependency* introduced by Cohen [34]. Informally, a program satisfies noninterference if the values of its public outputs do not depend on the values of its secret inputs. Formulated in terms of program executions, if the program is run with different secret inputs, while holding the public values fixed, the public output must not change.

2.1.2. General noninterference

We will show how all four variants of noninterference mentioned above — termination-insensitive, termination-sensitive, progress-insensitive and progress-sensitive noninterference — can be seen as instances of a general schema, which closely follows the intuition that the public output should be independent of the secret input.

Let c range over the commands of some language, and let $\langle c, E\rangle \Downarrow o$ be an *evaluation relation* read c executes in the environment E producing some observable behavior o. Assume further a *low equivalence* relation, $\sim$, on environments that captures that the public parts of the environments are equal, and an *indistinguishability relation*, $\simeq$, on behavior that captures the notion of *attacker indistinguishability*, i.e., if $o_1 \simeq o_2$ then the attacker is not able to distinguish between o_1 and o_2. In the following, we will use observable behavior of a program and the behavior of a program interchangeably.

A program c satisfies *noninterference*, written $NI(c)$, if, for any two low equivalent environments, a successful run of the program in one of the environments with behavior o_1 guarantees a successful run in the other environment with behavior some o_2 such that the behaviors are indistinguishable.

$$NI(c) = \forall E_1, E_2 \,.\, E_1 \sim E_2 \wedge \langle c, E_1\rangle \Downarrow o_1 \Longrightarrow \exists o_2 \,.\, \langle c, E_2\rangle \Downarrow o_2 \wedge o_1 \simeq o_2$$

If we chose the produced behavior to be the public output of the program and chose the indistinguishability relation to be equality, $NI(c)$ captures the intuition of noninterference above precisely, i.e., that the public output should remain equal when run with different secret input while retaining the public input equal.

Below we show how the three parts — 1) low equivalence, 2) the evaluation relation (in particular what is considerer observable behavior), and 3) observational indistinguishability — of this general schema can be concretely instantiated with respect to two different languages with different semantics (one big-step and one small-step) to achieve the four notions of noninterference: termination-insensitive, termination-sensitive, progress-insensitive, and progress-sensitive noninterference.

2.2. Termination-insensitive vs. termination-sensitive noninterference

To be concrete, we investigate termination-insensitive and termination-sensitive noninterference in terms of a minimal batch-job while language. A batch-job program is a program that starts in an environment and produces a new environment or diverges. The attacker model of the batch-job setting is that the attacker is able to inspect the public part of the final environment.

$$\frac{}{\langle skip, E\rangle \Rightarrow E} \qquad \frac{\langle c_1, E_1\rangle \Rightarrow E_2 \quad \langle c_2, E_2\rangle \Rightarrow E_3}{\langle c_1; c_2, E_1\rangle \Rightarrow E_3}$$

$$\frac{\langle e, E_1\rangle \Rightarrow true \quad \langle c_1, E_1\rangle \Rightarrow E_2}{\langle \textit{if } e \textit{ then } c_1 \textit{ else } c_2, E_1\rangle \Rightarrow E_2} \qquad \frac{\langle e, E_1\rangle \Rightarrow false \quad \langle c_2, E_1\rangle \Rightarrow E_2}{\langle \textit{if } e \textit{ then } c_1 \textit{ else } c_2, E_1\rangle \Rightarrow E_2}$$

$$\frac{\langle e, E_1\rangle \Rightarrow true \quad \langle c, E_1\rangle \Rightarrow E_2 \quad \langle \textit{while } e \textit{ do } c \textit{ done}, E_2\rangle \Rightarrow E_3}{\langle \textit{while } e \textit{ do } c \textit{ done}, E_1\rangle \Rightarrow E_3}$$

$$\frac{\langle e, E\rangle \Rightarrow false}{\langle \textit{while } e \textit{ do } c \textit{ done}, E\rangle \Rightarrow E} \qquad \frac{\langle e, E\rangle \Rightarrow v}{\langle x := e, E\rangle \Rightarrow E[x \mapsto v]}$$

Table 1. Big-step Semantics

2.2.1. Batch-job while language

Assume a standard expression, ranged over by e, language consisting of integer and boolean literals, variables, and the standard boolean and arithmetic operators and let the commands, ranged over by c, be built up by skip, sequence, conditional branches, iteration, and assignment.

$$e ::= b \mid n \mid x \mid e_1 \star e_2$$
$$c ::= skip \mid c_1;\ c_2 \mid \textit{if } e \textit{ then } c_1 \textit{ else } c_2 \mid \textit{while } e \textit{ do } c \textit{ done} \mid x := e$$

Semantics The batch-job semantics is defined as a big-step operational semantics. Let the values, ranged over by v, be the integers and the booleans, and let the *variable environments*, ranged over by γ, be maps from variable names to values. Let the *environments*, ranged over by E, consist of a single *variable environment* — the section on progress insensitive and progress sensitive noninterference below extends the environments to model input and output. Let $E[x]$ denote looking up variable x in the variable environment of E, and $E[x \mapsto v]$ denote updating variable x with value v in E resulting in a new environment such that $E[x \mapsto v][x] = v$, and $E[x \mapsto v][y] = E[y]$ for $y \neq x$.

Assume an evaluation relation for expressions, $\langle e, E\rangle \Rightarrow v$, read the expression e executes in the environment E yielding the value v, defined structurally as follows.

$$\langle n, E\rangle \Rightarrow n \quad \langle b, E\rangle \Rightarrow b \quad \frac{E[x] = v}{\langle x, E\rangle \Rightarrow v} \qquad \frac{\langle e_1, E\rangle \Rightarrow v_1 \quad \langle e_2, E\rangle \Rightarrow v_2}{\langle e_1 \star e_2, E\rangle \Rightarrow v_1 \star v_2}$$

Similarly we define an evaluation relation for commands, $\langle c, E_1\rangle \Rightarrow E_2$, read the command c executes in environment E_1 and terminates yielding the new environment E_2. The big-step semantics is defined in Table 1.

2.2.2. Security policy language for the batch-job while language

Frequently, the policy language for confidentiality is in terms of a lattice of *security levels* [38,52], where the lattice describes how information may flow between the different levels. For instance, a typical security lattice — and the lattice used in this document — is the two-point lattice described by $H > L$, which introduces two classifications of information — either secret or public — and expresses that secret information must never be considered public, but accepts flows the other way.

Let $\sigma ::= H \mid L$ be security labels corresponding to the security levels and let Γ range over *variable security maps*, maps from variable names to security labels. The

intuition is that variables mapped to L should only contain public information, whereas variables mapped to H may contain secrets. In the following, we refer the former as public variables and the latter as secret variables.

In the examples below it is understood that h is a secret variable, and that l is a public variable, i.e., the examples should be read with respect to the following variable security map $\Gamma = \{h \mapsto H, l \mapsto L\}$.

2.2.3. Low equivalence

Based on the variable maps we can define low equivalence of environments structurally by demanding that the parts labeled public by the map are equal. Technically, low equivalence is formulated as a family of relations indexed by variable security maps.

$$\frac{\sigma = L \Longrightarrow v_1 = v_2}{v_1 \sim_\sigma v_2} \qquad \frac{\forall x \in dom(\Gamma) \,.\, \gamma_1(x) \sim_{\Gamma(x)} \gamma_2(x)}{\gamma_1 \sim_\Gamma \gamma_2}$$

Two values are low equivalent with respect to public is they are equal; any two values are low equivalent with respect to secret. Two variable environments are low equivalent with respect to a variable map Γ if for all variables in Γ the respective values in the variable environments are low equivalent with respect to the label given by Γ. Thus, two low equivalent variable environments are guaranteed to contain the variables labeled by the variable map, with the additional property that all variables labeled public will contain equal values. Finally, two environments are low equivalent with respect to a variable map if their variable environments are low equivalent with respect to the variabel map.

2.2.4. Termination-insensitive vs. termination-sensitive noninterference

In the batch-job setting it is reasonable to assume an attacker model where the attacker is able to inspect the public parts of the final environment, as indicated by a variable map that maps the variables the attacker can inspect to L.

By letting the final environments of programs be the observable behavior, and let the basis of the indistinguishable relation be low equivalence, we capture the intuition that the public parts of the final environments be independent on secrets. In addition, we let the *divergence* of programs be observable behavior. Making divergence observable behavior is immediate, and does not require explicit modeling of divergence, since, in the semantics of Table 1, failure to execute implies divergence, i.e. a program c diverges in E_1 if $\neg\exists E_2 \,.\, \langle c, E_1\rangle \Rightarrow E_2$.

By letting divergence be an observable behavior, the difference between termination insensitivity and termination sensitivity can be expressed by the indistinguishability relation as shown below.

Let the observable behavior, ranged over by o, be defined by the environments E together with the symbol $\diamond$, which indicates divergence. The evaluation relation for termination-insensitive and termination-sensitive noninterference can be formulated as follows.

$$\frac{\langle c, E_1\rangle \Rightarrow E_2}{\langle c_1, E_1\rangle \Downarrow E_2} \qquad \frac{\neg\exists E_2 \,.\, \langle c, E_1\rangle \Rightarrow E_2}{\langle c_1, E_1\rangle \Downarrow \diamond}$$

If the program terminates the final environment is the observable behavior; otherwise, the program diverges.

Termination-Insensitive noninterference (TINI) TINI [116,98] demands the following property. For two low equivalent environments, if the program terminates in both producing two new environments, then the environments must be low equivalent. As indicated above, we capture this intuition by letting divergence be indistinguishable from everything else, and two environments be indistinguishable if they are low equivalent.

Two observations are termination-insensitive (TI) indistinguishable if 1) they are low equivalent environments, or 2) either is divergence.

$$\frac{}{\diamond \simeq_{TI} o_2} \qquad \frac{}{o_1 \simeq_{TI} \diamond} \qquad \frac{E_1 \sim_\Gamma E_2}{E_1 \simeq_{TI} E_2}$$

A program c satisfies termination-insensitive noninterference if pairwise execution in low equivalent environments results in TI-indistinguishable behavior, i.e., low equivalent environments in the case both programs terminate.

$$TINI(c) = \forall E_1, E_2 \,.\, E_1 \sim_\Gamma E_2 \wedge \langle c, E_1\rangle \Downarrow o_1 \Longrightarrow \exists o_2 \,.\, \langle c, E_2\rangle \Downarrow o_2 \wedge o_1 \simeq_{TI} o_2$$

This way, if either of the programs diverges, nothing is demanded, whereas if both programs terminate, we demand that the resulting environments are low equivalent, which corresponds exactly to the description of TINI above.

For batch-job programs this is a perfectly adequate formulation of security; it does not offer complete information security, but is limited to leaking of 1 bit per program run. Consider the following program that loops if the public value l is equal to the secret value h.

```
if (l == h) then while (true) do skip done
```

From the termination behavior of the program an observer can deduce whether $l == h$ or not corresponding to 1 bit of information.

Termination-sensitive noninterference (TSNI) TSNI [113,98] demands the following property. For two low equivalent environments, if the program terminates in one, then the program must terminate in the other as well and the resulting environments must be low equivalent.

To capture TSNI it suffices to change the indistinguishability relation of TINI in such a way that divergence is not considered indistinguishable from normal termination. Two environments are termination-sensitive (TS) indistinguishable if 1) they are low equivalent environments, or 2) they are both divergence.

$$\frac{}{\diamond \simeq_{TS} \diamond} \qquad \frac{E_1 \sim_\Gamma E_2}{E_1 \simeq_{TS} E_2}$$

A program c satisfies termination-insensitive noninterference if pairwise execution in low equivalent environments results in TS-indistinguishable behavior, i.e., both terminate with low equivalent environments, or both diverge.

$$TSNI(c) = \forall E_1, E_2 \,.\, E_1 \sim_\Gamma E_2 \wedge \langle c, E_1\rangle \Downarrow o_1 \Longrightarrow \exists o_2 \,.\, \langle c, E_2\rangle \Downarrow o_2 \wedge o_1 \simeq_{TS} o_2$$

2.2.5. *Classical formulation of batch-job noninterference*

Rather than the formulation above, the classical formulation of batch-job noninterference is in terms of the preservation of low equivalence under execution. For any two low equivalent environments, if the program terminates in both environments then the resulting environments should be low equivalent.

$$TINI_\Gamma(c) = \forall E_{11}, E_{12} \,.\, E_{11} \sim_\Gamma E_{12} \wedge \langle c, E_{11}\rangle \Rightarrow E_{21} \wedge \langle c, E_{12}\rangle \Rightarrow E_{22} \Longrightarrow E_{21} \sim_\Gamma E_{22}$$

A minor change to the security definition makes it termination sensitive. Instead of demanding that if the program terminates in two low equivalent environments then the resulting environments are low equivalent, we demand that for any low equivalent environments if the program terminates in one, the it must terminate in the other as well, and the resulting environments should be low equivalent.

$$\begin{aligned} TSNI_\Gamma(c) = \forall E_{11}, E_{12} \,.\, & E_{11} \sim_\Gamma E_{12} \wedge \\ & \langle c, E_{11}\rangle \Rightarrow E_{21} \Longrightarrow \exists E_{22} \,.\, \langle c, E_{12}\rangle \Rightarrow E_{22} \wedge E_{21} \sim_\Gamma E_{22} \end{aligned}$$

It is easy to verify that these definitions are equivalent to the formulations in the previous section.

2.3. *From termination to progress*

TINI and TSNI are not adequate for programs where the attacker can inspect intermediate steps of the computation, e.g., in the presence of input or output. For such languages TINI and TSNI open up for the entire secret to be leaked instantly. The reason for this is that leaks can be hidden by non-terminating computations. Consider the following program, which is trivially secure with respect to both TINI and TSNI, since it does not terminate for any inputs.

```
output secret on public_channel; while true do skip done
```

Hence, there is need to explicitly model intermediate observations in security definitions for interactive programs. This brings us to progress-insensitive and progress-sensitive noninterference.

2.4. *Progress-insensitive vs. progress-sensitive noninterference*

We investigate progress-insensitive and progress-sensitive noninterference in terms of a minimal while language with input and output. The assumption is that the attacker is able to inspect the public output of the program.

2.5. *While language with input and output*

To model input and output, two commands are added to the language: $in_\sigma\ x$ which reads a value and stores it in x, and $out_\sigma\ x$ which outputs the value of x. The input and output commands are indexed by security labels. The intention is that in_L models sources of

$$\frac{}{\langle skip, E\rangle \to E}$$

$$\frac{\langle c_1, E_1\rangle \xrightarrow{\dot{v}} \langle c_1', E_2\rangle}{\langle c_1; c_2, E_1\rangle \xrightarrow{\dot{v}} \langle c_1'; c_2, E_2\rangle} \qquad \frac{\langle c_1, E_1\rangle \xrightarrow{\dot{v}} E_2}{\langle c_1; c_2, E_1\rangle \xrightarrow{\dot{v}} \langle c_2, E_2\rangle}$$

$$\frac{\langle e, E\rangle \Rightarrow true}{\langle if\ e\ then\ c_1\ else\ c_2, E\rangle \to \langle c_1, E\rangle} \qquad \frac{\langle e, E\rangle \Rightarrow false}{\langle if\ e\ then\ c_1\ else\ c_2, E\rangle \to \langle c_2, E\rangle}$$

$$\frac{\langle e, E\rangle \Rightarrow true}{\langle while\ e\ do\ c\ done, E\rangle \to \langle c; while\ e\ do\ c\ done, E\rangle}$$

$$\frac{\langle e, E\rangle \Rightarrow false}{\langle while\ e\ do\ c\ done, E\rangle \to E} \qquad \frac{\langle e, E\rangle \Rightarrow v}{\langle x := e, E\rangle \to E[x \mapsto v]}$$

$$\frac{\iota_1 \downarrow_\sigma (v, \iota_2)}{\langle in_\sigma\ x, (\gamma, \iota_1, \omega)\rangle \to (\gamma[x \mapsto v], \iota_2, \omega)}$$

$$\frac{\gamma[x] = v \qquad (v, \omega_1) \uparrow_L \omega_2}{\langle out_L\ x, (\gamma, \iota, \omega_1)\rangle \xrightarrow{v} (\gamma, \iota, \omega_2)} \qquad \frac{\gamma[x] = v \qquad (v, \omega_1) \uparrow_H \omega_2}{\langle out_H\ x, (\gamma, \iota, \omega_1)\rangle \to (\gamma, \iota, \omega_2)}$$

Table 2. Small-step semantics

public information, whereas in_H models sources of secret information. Similarly, out_L models public sinks, i.e., sinks that the attacker can inspect, and out_H models secret sinks.

$$c ::= skip \mid c_1;\ c_2 \mid if\ e\ then\ c_1\ else\ c_2 \mid while\ e\ do\ c\ done \mid x := e \mid in_\sigma\ x \mid out_\sigma\ x$$

Semantics The semantics is defined as a small-step operational semantics, in order to allow for the differentiation between infinitely producing programs and silent divergence. In addition, computation steps that produce attacker observable behavior are decorated with a representation of that behavior — in this case the values output using out_L.

The environments, E, are extended to triples, (γ, ι, ω), where ι, and ω range over pairs of lists of values (one for each security level) representing the available input to the program, and the output of the program, respectively.

Let the pair $\langle c, E\rangle$ of a command and an environment be an *evaluation context*, and let $\dot{v}$ be a decoration, denoting v or nothing. A *terminating* transition, $\langle c, E_1\rangle \xrightarrow{\dot{v}} E_2$, represents evaluation that terminates in one step and is read the command c terminates in one step when evaluated in environment E_1 yielding environment E_2 and the possible observable behavior $\dot{v}$. A *non-terminating* transition, $\langle c_1, E_1\rangle \xrightarrow{\dot{v}} \langle c_2, E_2\rangle$, represents evaluation that does not terminate in one step of evaluation and is read the evaluation context $\langle c_1, E_1\rangle$ yields in one step of evaluation the new evaluation context $\langle c_2, E_2\rangle$ and the possible observable behavior $\dot{v}$. The small-step semantics is defined in Table 2, where $(v \cdot \overline{v}_L, \overline{v}_H) \downarrow_L (v, (\overline{v}_L, \overline{v}_H))$, $(\overline{v}_L, v \cdot \overline{v}_H) \downarrow_H (v, (\overline{v}_L, \overline{v}_H))$, $(v, (\overline{v}_L, \overline{v}_H)) \uparrow_L (v \cdot \overline{v}_L, \overline{v}_H)$, and $(v, (\overline{v}_L, \overline{v}_H)) \uparrow_H (\overline{v}_L, v \cdot \overline{v}_H)$ provide shorthand notation for public and secret input and output.

2.5.1. Low equivalence

For values and variable environments the low equivalence relation remains the same as defined for TINI and TSNI above; for environments it must be extended to deal with the input and output lists.

$$\frac{\sigma = L \Longrightarrow v_1 = v_2}{v_1 \sim_\sigma v_2} \qquad \frac{\forall x \in dom(\Gamma)\ .\ \gamma_1(x) \sim_{\Gamma(x)} \gamma_2(x)}{\gamma_1 \sim_\Gamma \gamma_2}$$

$$(\overline{v}, \overline{v}_1) \sim (\overline{v}, \overline{v}_2) \qquad \frac{\gamma_1 \sim_\Gamma \gamma_2 \qquad \iota_1 \sim \iota_2 \qquad \omega_1 \sim \omega_2}{(\gamma_1, \iota_1, \omega_1) \sim_\Gamma (\gamma_2, \iota_2, \omega_2)}$$

Two environments are low equivalent given with respect to a variable map if the variable environments are low equivalent with respect to the variable map and the input and output are low equivalent, i.e., that the public input and output lists are equal.

Technically, we could remove the demand that the public output lists be equal, since, in the setting below, it is implied by the demand of equality of public observables.

2.5.2. Progress-insensitive vs. progress-sensitive noninterference

Let R range over execution contexts, $\langle c, E\rangle$, and environments, E. We define $\langle c, E\rangle \overset{\dot{v}}{\Rightarrow} R$ to capture evaluation until observable output or termination with or without observable output.

$$\frac{\langle c_1, E_1\rangle \rightarrow^* \langle c_2, E_2\rangle \qquad \langle c_2, E_2\rangle \overset{v}{\rightarrow} R}{\langle c_1, E_1\rangle \overset{v}{\Rightarrow} R} \qquad \frac{\langle c, E_1\rangle \rightarrow^* E_2}{\langle c, E_1\rangle \Rightarrow E_2}$$

Let vs range over lists of values with $\cdot$ denoting the cons operation for lists. We extend the above relation to capture evaluation with zero or more observables as follows.

$$\frac{}{\langle c, E\rangle \Rightarrow \langle c, E\rangle} \qquad \frac{\langle c_1, E_1\rangle \overset{v}{\Rightarrow} \langle c_2, E_2\rangle \qquad \langle c_1, E_1\rangle \overset{vs}{\Rightarrow} \langle c_2, E_2\rangle}{\langle c_1, E_1\rangle \overset{v\cdot vs}{\Rightarrow} \langle c_2, E_2\rangle}$$

Note that $\neg\exists R, vs\ .\ \langle c, E\rangle \overset{vs}{\Rightarrow} R$ implies that $\langle c, E\rangle$ diverges silently. With this we can formulate the evaluation relation for progress-insensitive and progress-sensitive noninterference.

$$\frac{\langle c, E\rangle \overset{vs}{\Rightarrow} R}{\langle c, E\rangle \Downarrow vs} \qquad \frac{\langle c_1, E_1\rangle \overset{vs}{\Rightarrow} \langle c_2, E_2\rangle \qquad \neg\exists R, vs'\ .\ \langle c_2, E_2\rangle \overset{vs'}{\Rightarrow} R}{\langle c, E\rangle \Downarrow vs \cdot \diamond}$$

Progress-insensitive noninterference (PINI) PINI [3,7,21] demands the following property. For two environments with equal public values, if the program can execute one step producing some observable behavior, then either the program in the other environment diverges silently or produces the same observable behavior. Further, the rest of the two respective executions preserves this stepwise property. This implies that the observable behavior of the program in different low equivalent environments is independent of secrets up to silent divergence.

In the following, let o range over list of values where the last element is allowed to be $\diamond$ to indicate silent divergence. Two lists of observable behavior are progress-insensitive (PI) indistinguishable if 1) they are equal, or 2) one is a divergence terminated prefix of the other.

$$o_1 \simeq_{PI} o_2 \overset{def}{=} o_1 = o_2 \ \vee (\exists o, o'\ .\ o_1 = o \cdot \diamond \wedge o_2 = o \cdot o') \ \vee (\exists o, o'\ .\ o_2 = o \cdot \diamond \wedge o_1 = o \cdot o')$$

A program satisfies progress-insensitive noninterference if pairwise execution in low equivalent environments results in PI-indistinguishable behavior, i.e., both executions produce the same output, or both produce the same output up to the point where one diverges.

$$PINI(c) = \forall E_1, E_2 \,.\, E_1 \sim_\Gamma E_2 \wedge \langle c, E_1\rangle \Downarrow o_1 \implies \exists o_2 \,.\, \langle c, E_2\rangle \Downarrow o_2 \wedge o_1 \simeq_{PI} o_2$$

Even though progress-insensitive noninterference closes the possibility of leaking the secret in linear time in the size of the secret it is susceptible to the possibility of *enumeration attacks*, where the entire secret is leaked to the attacker. Consider the following program

```
i := 0;
while i < maxNat do
  out_L i;
  if (i == secret) then (while true do skip done)
                   else skip;
  i:=i+1
done
```

Askarov et al. [3] show in the sequential setting that, for uniformly distributed secrets, the advantage of a polynomial-time attacker to learn the secret is negligible in the size of the secret, making PINI appropriate for large uniformly distributed secrets such as cryptographic keys. It is important to note, however, that concurrency empowers such attacks to leak the secret in linear time in the size of the secret.

Progress-sensitive noninterference (PSNI) Similar to above, demanding that if one execution makes *progress*, i.e., takes one step of execution with observable behavior, then so does the other in addition to the above demands on stepwise matching of the public observables. This corresponds to progress sensitive noninterference, since it forces the progress of the program to be independent of secrets. PSNI subsumes TSNI for interactive programs, since nontermination is formally equal to the absence of progress.

Two sequences of observations are progress-sensitive (PS) indistinguishable if they are equal, i.e., we remove the possibility of one being a divergence terminate prefix of the other, thus demanding that if one diverges then so must the other.

$$o_1 \simeq o_2 \stackrel{def}{=} o_1 = o_2$$

A program satisfies progress-sensitive noninterference if pairwise execution in low-equivalent environments results in PS-indistinguishable behavior, i.e., they both produce the same sequence of output, before possibly both diverging.

$$PINI(c) = \forall E_1, E_2 \,.\, E_1 \sim_\Gamma E_2 \wedge \langle c, E_1\rangle \Downarrow o_1 \implies \exists o_2 \,.\, \langle c, E_2\rangle \Downarrow o_2 \wedge o_1 \simeq_{PS} o_2$$

2.6. A note on compositionality and flow sensitivity

Secure composition allows for building secure components by combining secure components, and is important for scalable security analysis regardless of the goal of the anal-

ysis, i.e., confidentiality, integrity or otherwise. Compositionality has been an important topic in the context of programming languages [68,104,75,101,97,71] — for instance, type systems are inherently compositional in that programs (statements and expression) are typed by typing the subparts.

However, compositionality is typically not an intrinsic property of security definitions. Rather, compositional security properties that imply the weaker non-compositional security property are frequently formulated to enable proofs of correctness of compositional enforcement methods.

2.6.1. *Compositionality of* TINI *and* TSNI

The formulations of *TINI* and *TSNI* above are compositional under the same variable map with respect to sequential composition. We exemplify with the classical formulation of the former.

$$TINI_\Gamma(c_1) \wedge TINI_\Gamma(c_2) \Longrightarrow TINI_\Gamma(c_1; c_2)$$

This fact is easy to see in the classical formulation, which is stated as the preservation of low equivalence of the environments under execution. We must show that $TINI_\Gamma(c_1; c_2)$, i.e

$$TINI_\Gamma(c_1; c_2) = \forall E_{11}, E_{12} \,.\, E_{11} \sim_\Gamma E_{12} \wedge \langle c_1; c_2, E_{11}\rangle \Rightarrow E_{21} \wedge \langle c_1; c_2, E_{12}\rangle \Rightarrow E_{22} \Longrightarrow E_{21} \sim_\Gamma E_{22}$$

Thus, assume that (1) $E_{11} \sim_\Gamma E_{12}$, (2) $\langle c_1; c_2, E_{11}\rangle \Rightarrow E_{21}$, and (3) $\langle c_1; c_2, E_{12}\rangle \Rightarrow E_{22}$. From (2) and (3) we have that there exists E'_{11}, and E'_{12} such that (4) $\langle c_1, E_{11}\rangle \Rightarrow E'_{11}$, (5) $\langle c_2, E'_{11}\rangle \Rightarrow E_{21}$, (6) $\langle c_1, E_{12}\rangle \Rightarrow E'_{12}$, and (7) $\langle c_2, E'_{12}\rangle \Rightarrow E_{22}$. From $TINI_\Gamma(c_1)$ together with (1), (4) and (6) we get that (8) $E'_{11} \sim_\Gamma E'_{12}$. Now, $TINI_\Gamma(c_2)$ together with (8), (5) and (7) allows us to conclude.

On the other hand, the formulation of *PINI* and *PSNI* are not compositional, since they offer no guarantees on the produced environments, only on the observable behavior. However, they can both be strengthened to become compositional by making the final environments observable behavior and demanding that they are low analogous to *TINI* and *TSNI*. In the following, let *CompPINI*, and *CompPSNI* denote compositional versions of *PINI* and *PSNI* obtained in this manner, respectively.

2.6.2. *Flow sensitivity*

Compositionality may come at the price of classifying some secure programs as insecure (from the perspective of attacker observations). Consider, for instance, the following program.

```
out_L l; l := h
```

While it is secure with respect to *PINI* and *PSNI*, i.e. with respect to attacker observations, when run on its own, it is insecure with respect to *CompPINI* and *CompPSNI*, since the resulting environments are not low equivalent with respect to the initial vari-

able map. Indeed, the program cannot be securely composed with all other programs — for instance, sequential composition of the program with itself would result in an insecure program — it is secure on its own and can be securely composed with any program considering l as secret.

A common way of weakening the demands while retaining compositionality is by allowing the initial and final variable maps to differ. This is known as *flow sensitivity*, since it allows the security map to change with the control flow. For brevity, we exemplify flow sensitivity in terms of the classical formulation of *TINI*; flow-sensitive weakenings of *CompPINI* and *CompPSNI* can be formed in a similar way.

$$TINI_{\Gamma_1,\Gamma_2}(c) = \forall E_{11}, E_{12} \,.\, E_{11} \sim_{\Gamma_1} E_{12} \wedge \langle c, E_{11}\rangle \Rightarrow E_{21} \wedge \langle c, E_{12}\rangle \Rightarrow E_{22} \Longrightarrow \wedge E_{21} \sim_{\Gamma_2} E_{22}$$

With this formulation of *TSNI* the above example is secure with respect to the initial variable map $\Gamma_1 = \{l \mapsto L, h \mapsto H\}$, and the final variable map $\Gamma_2 = \{l \mapsto H, h \mapsto H\}$. This map also allows for sequential composition with any command secure with respect to Γ_2 as initial variable map. In general, the following holds for sequential composition.

$$TINI_{\Gamma_1,\Gamma_2}(c_1) \wedge TINI_{\Gamma_2,\Gamma_3}(c_2) \Longrightarrow TINI_{\Gamma_1,\Gamma_3}(c_1; c_2)$$

The correctness argument is analogous to the one above.

Further strengthenings of the compositionality result can be obtained. For instance, consider not demanding equality on the intermediate variable map, but rather variable map inclusion $\supseteq$,

$$\Gamma_1 \supseteq \Gamma_2 \Longleftrightarrow \forall x \in dom(\Gamma_2) \,.\, \Gamma_1(x) = \Gamma_2(x)$$

where $\Gamma_1 \supseteq \Gamma_2$ should be read as Γ_1 includes Γ_2 in the sense that environments that are low equivalent with respect to Γ_1, are also low equivalent with respect to Γ_2, i.e.,

$$E_1 \sim_{\Gamma_1} E_2 \wedge \Gamma_1 \supseteq \Gamma_2 \Longrightarrow E_1 \sim_{\Gamma_2} E_2$$

With this we can formulate that, for the sequence $c_1; c_2$, the demands by c_2 are included in guarantees of c_1, expressed as follows.

$$TINI_{\Gamma_1,\Gamma_2}(c_1) \wedge \Gamma_2 \supseteq \Gamma_3 \wedge TINI_{\Gamma_3,\Gamma_4}(c_2) \Longrightarrow TINI_{\Gamma_1,\Gamma_4}(c_1; c_2)$$

Again, the correctness argument is analogous.

2.7. Information flow and concurrency

Concurrency poses an important challenge for information flow security. In addition to forcing progress sensitivity, parallel composition of secure programs is not necessarily secure. One reason for this is *internal timing* channels. Consider the following program in which both sub-programs satisfies noninterference, but when run in parallel they leak the secret h to the public variable l.

```
if h {sleep(100)}; l = 1 | sleep(50); l = 0
```

Smith and Volpano [108] investigate a notion of possibilistic noninterference. Possibilistic noninterference states that the possible low outputs of the program should not vary as high inputs are varied. However, possibilistic noninterference is only meaningful for a very restricted set of probabilistic schedulers. Sabelfeld and Sands [104] consider a form of probabilistic noninterference for a language with dynamic thread creation. They show how to define security for a wide class of schedulers, not excluding deterministic schedulers. Mantel and Sabelfeld [70] investigate a timing-sensitive security property for multithreaded programs, later extended to the distributed setting [71]. Sabelfeld [96] considers bisimulation based formulations of confidentiality for multi-threaded programs, focusing on formulations for timing- and probability-sensitive confidentiality. Sabelfeld derives relationships between scheduler specific, scheduler independent and strong confidentiality. Roscoe [87] investigates confidentiality properties in a process-calculus setting. A notion *low-view determinism* is presented, which demands that abstracted publicly observable results are deterministic and, thus, independent of secret inputs.

2.8. Information flow and interactive/reactive programs

Clark and Hunt investigate noninterference for interactive programs [31] and find that for *deterministic* interactive programs attacker *strategies* — functions that produce new input based on the previous communication history [81] — do not have to be employed. Castellani et al. [74] study noninterferences for a class of synchronous reactive programs, introducing a concept of *instant* to handle the absence of signals. Bohannon et al. [21] explore a spectrum of different definitions of secure information flow for reactive programs, in the setting of an event driven sequential language. In continued work, Bohannon and Pierce [20] formalize the core functionality of a Web browser in a model called Featherweight Firefox.

2.9. Erasure

The notion of information erasure is related to information flow. For instance, cryptographic devices might be required to erase secret keys once they are done using them, or an online retailer might be obliged to erase customer data after the transaction is done.

Chong and Myers [27] investigate the semantics of policies for information erasure in the setting of a non-interactive language, but without giving a method of enforcing the policies. Hunt and Sands [50] show that erasure policies can be encoded as flow-sensitive noninterference in the setting of an interactive language, and propose a type system enforcing the policies. Del Tedesco et al. [36] show how ideas from dynamic taint analysis can be used to track sensitive data through a program and provide on-demand erasure.

2.10. Declassification

For many applications a complete separation between secret and public is too restrictive. Consider for instance the login screen of an operating system — when a user tries logging in the response of the system gives away information about the password. If access is refused we know that the attempted password was not the correct one. Even though

this gives away partial information about the password, we deem this secure. Another important class of examples is data aggregation. Consider for instance a program that computes average salaries — even though each individual salary may be secret we might want to be able to publish the average.

Clearly, we need a way to *declassify* information, i.e., lowering the security classification of selected information. Sabelfeld and Sands [102] identify four different dimensions of declassification, *what* is declassified, *who* is able to declassify, *where* the declassification occurs and *when* the declassification takes place.

what As illustrated above, it is important to be able to specify what information is declassified, e.g., the four last digits of a credit card number, the average salary. Policies for partial release must guarantee an upper limit on what information is released.

who Another important aspect is who controls the release of information. This pertains to information *integrity* — if the attacker is able to control what information is declassified he might be able to mount a *laundry* attack, i.e., unintended leaks hidden by the systems declassification policy.

where Sabelfeld and Sands identify two principal forms of release locality. Related to the *what* and *when* dimension, the *where* dimension is the most immediate interpretation of where in terms of code locality. The other form is level locality, describing where information may flow relative to the security levels of the system.

when The temporal dimension of declassification pertains to when information is leaked. Sabelfeld and Sands identify three classes of temporal release classifications, *Time-complexity based*, *Probabilistic* and *Relative*. The two former are related. Time-complexity based states that information will not be released until, at the earliest, after a certain time; typically as an asymptotic notion relative to the size of the secret. With probabilistic considerations one can talk about the probability of a leak being very small. The class of relative temporal policies are on the other hand related to program correctness. It controls when declassification can occur relative to other (possibly abstract) events in the system. For example: "downgrading of a software key may occur after confirmation of payment has been received."

Works that address the what *dimension* Lowe [61] investigates a quantifying information flow, where the capacity of channels is formulated in terms of the number of different behaviors of a high level user that can be distinguished by a low level user. Clark et al. [33,32] investigate an information theoretic approach to bounding the interference in a while language with iteration.Sabelfeld and Myers [99] introduce the notion of syntactic escape hatches to delimit the amount of information released. An escape hatch formed by an expression and, semantically, the information allowed to be released is characterized by the expression interpreted in the initial memory. Li and Zdancewic [58] present the concept of relaxed noninterference that mainly addresses the *what* dimension and, to a lesser extent, the *where* dimension. Giacobazzi and Mastroeni [44,45] introduce abstract noninterference, a parameterization of noninterference over an abstract interpretation modeling the power of the attacker. Di Pierro et al. [41] consider a quantitative approach to information flow and declassification in PCCP. Information leakage is allowed via a notion of process similarity, and the quantitative measure is related to the number of statistical tests needed to distinguish between process behaviors.

Works that address the who *dimension* The *who* dimension of declassification has been investigated in the context of *robustness* [78,4], which controls on whose behalf declassification may occur. Lux and Mantel [62] investigate a bisimulation-based condition that helps expressing who (or, more precisely, what input channels) may affect declassification. Myers and Liskov [76] model ownership in the decentralized label model, where declassification is considered safe if it is performed by the owner of the information.

Works that address the where *dimension* Matos and Boudol [73] introduce the concept of *non-disclosure* in the setting of concurrent ML. They introduce a local flow policy that allows the computation in the scope of such a declaration to implement information flow according to the local policy. Mantel and Sands [72] investigate the use of *intransitive noninterference* to control where in the classification lattice declassification is allowed — in combination with a syntactic construction for declassification they address both aspects of *where*. Intransitive noninterference [88,85,89,67] allows for policies where information may flow indirectly between two security levels, but not directly.

Works that address the when *dimension* Time-complexity based information declassification prevents the information from being released until, at the earliest, after a certain time.

Volpano and Smith [115] introduce the notion of relative secrecy. The idea is that the attacker cannot learn the secret in polynomial time. DiPierro et al. [41] consider a purely probabilistic notion of approximate noninterference, where a system is considered secure if the chance of an attacker making distinctions is smaller than some constant ϵ. Chong and Myers [26] investigate security policies, expressed in a logic form, which address when information is released. Recently, Magazinius et al. [64] have proposed support for decentralized policies with possible mutual distrust for tracking information flow in mashups. Their model of *composite delimited release* guarantees that a piece of data may be released only if all origins that own the data agree that it can be released.

Multidimensional declassification Mantel and Reinhard [69] address the *what* and *where* dimensions of declassification. Their security condition for *where* combines both code locality and where in the lattice — the latter in similar spirit to intransitive noninterference. In addition they provide two different escape hatch based security conditions for *what*. Askarov and Sabelfeld [6] consider the *what* and *where* dimension of information release policies. They present a system based on delimited release which achieves the *where* dimension by using accumulated escape hatches. Banerjee et al. [11] investigate an expressive policy language for *when*, *what* and *where* policies. In addition, Banerjee et al. [13] offer an abstraction based declassification that represents the declassification policy as an abstraction of the secret inputs. A program that satisfies the policy is guaranteed not to expose distinctions within a partition.

Knowledge-based formulations Recently a knowledge-based formulation has gained popularity, e.g., [42,5,24,22,23,7]. The attacker knowledge given a certain observable behavior is the set of all initial memories that produces the same observable behavior.

Knowledge-based information-flow security supersedes noninterference; both PINI and PSNI have natural interpretations in terms of knowledge. By defining *progress knowledge* — the knowledge the attacker gains by observing progress — PINI can be formulated by demanding that the knowledge of the attacker minus the progress knowledge remains constant under execution. Even easier is PSNI which is achieved by demanding

that attacker knowledge does not change during execution. The original work [5] focused on the *where* dimension of declassification but has late been extended to control the *what* dimension by applying the ideas of delimited release.

3. Integrity

Where confidentiality is a relatively well understood concept, what is meant by *integrity* is still partly unexplored. It is clear that there are different *facets* of integrity. Birgirsson et al. [19] identify different facets of integrity.

As Dual to Confidentiality Integrity in the area of information flow often means that trusted output is independent from untrusted input [18]. This is dual to the classical models of confidentiality, where public output is required to be independent from secret input.

As Generalized Invariants and Program Correctness Integrity in the area of access control [105] is concerned with improper/unauthorized data modification. The focus is on preventing data modification operations, when no modification rights are granted to a given principal. Integrity in the context of fault-tolerant systems is concerned with preservation of actual data. For example, a desired property for a file transfer protocol on a lossy channel is that the integrity of a transmitted file is preserved, i.e., the information at both ends of communication must be identical (which can be enforced by detecting and repairing possible file corruption). Integrity in the context of databases often means preservation of some important invariants, such as consistency of data and uniqueness of database keys. The list of different interpretations of integrity can be continued, including rather general notions as integrity as expectation of data quality and integrity as guarantee of accurate data and meaningful data [105,83]. Seeking to clarify the area of integrity policies, Li et al. [55] suggest a classification for data integrity policies into information flow, data invariant and program correctness policies. In a similar spirit, Guttman [47] identifies causality and invariance policies as two major types of data integrity policies. Furthermore, Birgirsson et al. argue that integrity via invariance is itself multi-faceted. For example, the literature (cf. [55]) features formalizations of invariance as predicate preservation (predicate invariance), which is not directly compatible with invariance of memory values (value invariance). Sabelfeld and Myers [98] observe that integrity has an important difference from confidentiality: a computing system can damage integrity without any external interaction, simply by computing data incorrectly. Thus, strong enforcement of integrity requires proving program correctness. Birgirsson et al. generalize the notion of invariants, so that it can describe predicate and value invariance, as well as program correctness.

Dimensions of confidentiality-dual integrity There are many situations where we might wish to upgrade the integrity levels of data. This is known as endorsement. When viewed as dual to confidentiality, the dimensions for declassification can also be applied to endorsement:

What The *what* dimension can be studied with essentially the same semantics and thus deals with what parts of information are endorsed. Interestingly, Li and Zdancewic [57] in a study of the dualization of relaxed noninterference [56], discuss some non-dual aspects of policies, stemming from whether the code itself is trusted or not.

When Temporal endorsement is common in certain scenarios. For example, if you choose to trust some low integrity data only after a digital signature has been verified. Other, complexity-theoretic notions are perhaps less natural in the integrity setting. Although one is able to say that, e.g., "low integrity data remains untrusted in any polynomial time computation", it is less obvious how this kind of property might be useful.

Where Both policy locality and code locality are natural for endorsement. For policy locality we may wish to ensure that untrusted data only becomes trusted by following a particular path (i.e., intransitive noninterference). From the point of view of code locality it is again natural to require that endorsement only takes place at the corresponding points in the program.

Who The *who* dimension is interesting because the notion already embodies a form of integrity. Robust declassification, for example, argues that low integrity data should not effect the decision of what gets declassified [77]. For integrity we might thus define a notion of robust endorsement to mean that the decision to endorse data should not itself be influenced by low integrity data. This approach can benefit from a non-dual treatment of endorsement. Because the potentially dangerous operations like declassification and endorsement are *privileged* operations, it might make sense to apply similar, not dual constrains.

Li et al. [55] discuss unifying policies for confidentiality and integrity in the context of the DLM. They offer a comparison between different models: the binary model, the write model, the trust model and the distrust model. Zdancewic and Myers [118] investigate the interaction between declassification and integrity. They introduce the concept of robust declassification, classification that cannot be influenced by the attacker. The demand is that declassification is only done in high integrity context. This prevents an attacker from laundering information. Myers et al. [77], present a generalization to robust declassification to include endorsement, thus giving a better account for the interaction between confidentiality and integrity. Askarov and Myers [4] introduce a semantic framework for declassification and endorsement. They investigate the power the attacker gains from declassification and introduce novel security conditions for *checked endorsements* and *robust integrity*. Chong and Myers [28] discuss an extension of robustness to systems with mutual distrust and show how the DLM can be used to characterize the power of an arbitrary attacker.

4. Information-flow enforcement

Historically, dynamic techniques are the pioneers of the area of information flow (e.g., [43]). They prevent explicit flows (as in *public* = *secret*) in program runs. In addition, they also address implicit flows (as in `if` *secret* `then` *public* = *1*) by enforcing a simple invariant of no public side effects in *secret contexts*, i.e., in the branches of con-

ditionals and loops with secret guards. These techniques, however, come without soundness arguments.

In their seminal work, Denning and Denning [39] suggest a static alternative for information-flow analysis. They argue that static analysis removes runtime overhead for security checks. This analysis prevents both explicit and implicit flows statically. The invariant of no public side effects in secret context is ensured by a syntactic check: no assignments to public variables are allowed in secret contexts. Denning and Denning do not discuss soundness, but Volpano et al. [116] show soundness by proving termination-insensitive noninterference, when they cast Denning and Denning's analysis as a security type system.

Later work is dominated by the use of static techniques for information flow [98]. The common wisdom appears to be that dynamic approaches are not a good match for security since monitoring a single path misses public side effects that could have happened in other paths.

In this light, it might be surprising that it is possible for purely dynamic enforcement to be *as secure as Denning-style static analysis* [100]. The key factor is termination. Denning-style static analysis are typically progress-insensitive (i.e., they ignore leaks via the termination behavior of the program). Thus, they satisfy termination-insensitive noninterference [116], which ignores the channel for signals via the termination of the program. If the monitor, by stopping the underlying program, can introduce nontermination, this feature can be used for collapsing information channels into the progress channel. The implicit-flow channel is one example: stopping the execution at an attempt of a public assignment in secret context is in fact sufficient for termination-sensitive security.

Progress-sensitive noninterference is attractive, but rather difficult to guarantee. Typically, strong restrictions (such as no loops with secret guards [113]) are enforced. Program errors exacerbate the problem. Even in languages like Agda [80], where it is impossible to write nonterminating programs, it is possible to write programs that terminate abnormally: for example, with stack overflow. Generally, abnormal termination due to resource exhaustion, is a channel for leaks that can be hard to counter.

The information flow tools Jif [79], FlowCaml [107] and the SPARK Examiner [14,25] avoid these problems by targeting termination-insensitive noninterference. The price is that the attacker may leak secrets by brute-force attacks via the termination channel. But there is formal assurance that these are the only possible attacks. Askarov et al. [3] show that if a program satisfies progress-insensitive noninterference, then the attacker may not learn the secret in polynomial running time in the size of the secret; and, for uniformly-distributed secrets, the probability of guessing the secret in polynomial running time is negligible. For small secrets this might not be satisfactory, calling for treating large and small secrets differently [37].

4.1. Static vs. dynamic enforcement

Static techniques have benefits of reducing runtime overhead and dynamic techniques have the benefits of permissiveness, which is of particular importance in dynamic applications, where freshly generated code is evaluated. This setting is becoming increasingly more important with the growing use of web browsers as application platforms, since the client side language, JavaScript, is a highly dynamic language.

First, JavaScript is dynamically typed, meaning that type checking is performed at runtime. This allows for a more liberal type system, at the expense of runtime overhead.

This also entails that programs may not exhibit static types, which forces dynamic or hybrid analyses.

Second, JavaScript allows the redefinition of functions, methods and prototypes — both user defined and built-in. This presents major challenges for information-flow security, since programs can be included by other programs. Thus, a program cannot assume to run in the standard environment — it may have been included by another program that has modified the standard environment, or it might include other programs that must be prohibited from doing so.

Third, the dynamic code evaluation feature of JavaScript provides a particular challenge. Dynamic code evaluation evaluates a given string using the `eval` function. Static analysis is bound to be conservative when analyzing programs that include `eval`, since the strings to be evaluated are typically not known at the time of analysis.

For example, it is not possible to statically determine if a program using `eval` is secure without being too conservative, since the parameter of `eval` might not be known at the time of analysis or might be subject to change. Moreover, in a heterogeneous environment as the Web, it is also difficult to assume properties about third-party scripts.

Another example to illustrate permissiveness of dynamic techniques is the program `if` $l < 0$ `then` $l = 1$ `else` $l = h$, where l and h are variables that store public and secret values, respectively. Static analysis, as traditional type systems [116], rejects this program as insecure due to the presence of the explicit flow $l = h$. In contrast, some dynamic techniques are able to accept executions of the program when $l < 0$ holds [100]. On the security side, however, both Denning-style analysis and dynamic enforcement have the same guarantees: progress-insensitive noninterference [100].

However, if progress-sensitive noninterference is desired, the absence of side effects of traces not taken becomes, indeed, hard to guarantee dynamically.

4.2. Static analysis

The predominant static technique for enforcing secure information flow statically is the use of type systems, e.g. [82,116,76,48,108,17,114,1,104,119,12,86].

In a security-typed language, the types of program variables and expressions are augmented with annotations that specify policies on the use of the typed data. Denning-style analyses prohibit implicit information flows by keeping track of the security level of the control, frequently known as the security level of the program counter and disallowing public side effects in secret contexts. This enforcement scheme is know as flow-insensitive, since it does not allow the security classification of program locations to vary.

In contrast, Hunt and Sands [49] investigate flow-sensitive type systems for a small while language. The type systems are parameterized over the choice of flow lattice for which the powerset lattice of program variables is shown to provide a principal typing. In addition Hunt and Sands show how to transform any flow-sensitive program into an equivalent program typable in a flow-insensitive type system. The concept of flow-sensitive information-flow security goes back to Banerjee and Amtoft [2], with the enforcement phrased as a Hoare logic rather than a type system.

Statically enforcing confidentiality With respect to practical implementations of information-flow security, Denning-style analyses form the core for information flow tools Jif [79], FlowCaml [107] and the SPARK Examiner [14,25]. Askarov and Sabelfeld

[8] present a novel treatment of secure exception handling. They allow secret exceptions to be uncaught given that they always are caught or always uncaught, and show that this is sound with respect to termination insensitive noninterference. Broberg and Sands investigate expressive dynamic information-flow policies — flow-locks — and present a type system for an ML like language [22], subsequently recast using a knowledge based definition [23] and extended to a role-based multi-principal settings [24].

Smith and Volpano [108] show how a strengthening of the type system for sequential programs [116] by disallowing looping on secrets is sufficient for ensuring security under purely nondeterministic schedulers. Sabelfeld [96] proposes a type based analysis for multi-threaded programs in the presence of synchronization. The type system excludes the possibility of synchronizing on secret data — directly or indirectly — in branches of high conditionals. Russo and Sabelfeld [91] present a type system that guarantees security for a wide class of schedulers for languages with dynamic threads. To achieve this the language is augmented with a pair of commands *hide* and *unhide* that move threads between different queues. By disallowing low threads to be scheduled while there are pending high threads internal timing leaks are prevented.

Castellani et al. [74] present a type system guaranteeing noninterference in a simple imperative reactive language. Bohannon et al. [21] explore different definitions of noninterference for reactive programs and define a simple reactive language with an information-flow type system to demonstrate the viability of the approach.

Matos and Boudol [73] introduce a local flow policy that allows the computation in the scope of such to implement information flow according to the local policy and design a type and effect system that enforces this policy. In addition by particularizing the case where the alternatives in a conditional branching both terminate Matos and Boudol show that typing of terminations leaks can be improved. Sabelfeld and Myers [99] introduce the notion of syntactic escape hatches to delimit the amount of information released. An escape hatch is formed by an expression and, semantically, the information allowed to be released is characterized by the expression interpreted in the initial memory. Sabelfeld and Myers present a type system for enforcing delimited release. The type system tracks the variables taking part in declassification and demands that those variables are not the target of update before the declassification statement. Askarov and Sabelfeld [6] extend the notion of delimited release with code locality and present a type system that enforces the new notion by disallowing declassification in secret contexts. Li and Zdancewic [58] present the concept of relaxed noninterference that mainly addresses the *what* dimension and to a lesser extent the *where* dimension enforced via a type system. The soundness theorem of the type system ensures that, if a program is well-typed, then there exists a proof of the security goal for the program. Mantel and Reinhard [69] present a security type system to enforce security with respect to the *what* and *where* dimensions of declassification, where their security condition for *where* combines both code locality and lattice locality.

Statically enforcing integrity Myers et al. [77] present a generalization to robust declassification to include endorsement, thus giving a better account for the interaction between confidentiality and integrity and present a type based enforcement. The enforcement is based on only high-integrity data being allowed to be declassified and that declassification might only occur in a high-integrity context. Chong and Myers [28] discuss an extension of robustness to systems with mutual distrust and present a type based enforcement. The type system relies on the fact that both the decision to declassify and

the information to be declassified are high-integrity and that the decision to endorse information must be of high integrity. Tripp et al. [109] present TAJ, a tool for scalable static taint analysis for Web applications. Being an industrial tool aimed at being able to handle existing complex Web applications no correctness argument is given.

4.3. Dynamic and hybrid analysis

One alternative is purely *dynamic* enforcement (e.g., [43,112,7,100]), that performs dynamic security checks similar to the ones enforced by static analysis. For example, an assignment is allowed by the monitor if the level of the assigned variable is high whenever there is a high variable on the right-hand side of the assignment (tracking explicit flows) or in case the assignment appears inside of a high conditional or while loop (tracking implicit flows). This mechanism dynamically keeps a simple invariant of no assignment to low variables in high context.

As previously noted it has been shown (e.g., [9,100,7,93]) that purely dynamic monitors can enforce the same security property as Denning-style static analysis: termination-insensitive noninterference. In addition, Sabelfeld and Russo [100] prove that sound purely dynamic information-flow enforcement is more permissive than static analysis in the *flow-insensitive* case (where variables are assigned security levels at the beginning of the execution and this assignment is kept unchanged during the execution).

Purely dynamic enforcement Shroff et al. [106] develop a monitor to track explicit and implicit information flow. The monitor is parameterized over a set of dependencies to track implicit flow. This set can either be computed statically or dynamically at the expense of possibly interfering runs while the monitor collects all dependencies. Russo and Sabelfeld [93] investigate securing timeout instructions in Web applications against internal timing leaks. They propose a monitor using a generalization of security contexts to include time. This way code snippets to be run at time t are run in the associated security context. Askarov and Sabelfeld [7] investigate dynamic tracking of policies for information release, or declassification, for a language with dynamic code evaluation and communication primitives. Russo and Sabelfeld [93] show how to secure programs with timeout instructions using execution monitoring. Furthermore, Russo et al. [94] investigate monitoring information flow in dynamic tree structures. Austin and Flanagan [9] present a dynamic analysis for secure information flow. They apply a no-sensitive-upgrade strategy — on an attempt to assign to a public variable in secret context, the public variable is marked as one that cannot be branched on later in the execution — to avoid the pitfalls of flow-sensitivity and dynamic enforcement. Austin and Flanagan [10] relaxe the no-sensitive-upgrade strategy to a permissive upgrade, where variables are allowed to be upgraded before the secret context, in which they are assigned to.

Hybrid enforcement Fusion of static and dynamic techniques is becoming increasingly popular [54,106,53,111]. These techniques offer benefits of increasing permissiveness because more information on the actual execution trace is available at runtime, while keeping runtime overhead moderate as some static information can be gathered before the execution. Russo and Sabelfeld [92] show formal underpinnings of the tradeoff between dynamism and permissiveness of flow-sensitive monitors. They also present a general framework for hybrid monitors that is parametric in the monitor's enforcement actions (blocking, outputting default values, and suppressing events). LeGuernic et al. [54]

consider a dynamic automaton based monitor for confidentiality. The monitor is a hybrid between dynamic and static enforcement; during execution abstractions of events are sent to the monitor which uses the abstraction to prohibit both explicit and implicit flows. LeGuernic [53] develops a hybrid monitor for concurrent programs. The monitor uses abstractions of program events, e.g., the modified variables of non-taken branches, the set of variables that must be locked for bodies of secret conditionals. Ligatti et al. [60] present a general framework for security policies that can be enforced by monitoring and modifying programs at runtime. They introduce edit automata that enable monitors to stop, suppress, and modify the behavior of programs. Chugh et al. [30] present a hybrid approach to handling dynamic execution. Their approach is staged into two stages. First, the information-flow properties for the available code is examined and a set of residual syntactic checks to be applied to the dynamic code is generated. Once the dynamic code is to be evaluated the residual checks are applied.

Code transformation In addition to execution monitors there exists a line of work that statically or dynamically filters, rewrites or wraps the code to enforce different properties [63]. Devriese and Piessens [40] consider enforcing noninterference by running multiple runs of the program, one for each security level. The idea is that you first perform the public computation, replacing any secret values with dummies. Thereafter you run the secret computation under certain restrictions. For a secure program this preserves the semantics of the original program and enforces noninterference. Phung et al. [84] consider an approach to modifying JavaScript code to become self protecting. The method is lightweight in the sense that it does not rely on browser modifications or runtime rewriting. Magazinius et al. [65] improve on [84] by removing a number of identified vulnerabilities and making the policy language more accessible to the policy writer. Chudnov and Naumann [29] presents a provably correct inlining of a dynamic flow-sensitive monitor. They prove security and transparency by connecting the inlined monitor with a VM monitor, known to have the desired security properties. Magazinius et al. [66] investigate on-the-fly inlining of a dynamic monitor to handle dynamic code evaluation. With respect to concurrency Barthe et al. [16] present a compositional transformation that closes internal timing channels in multithreaded programs with semaphores. Jang et al. [51] consider a rewriting based technique and tool for finding insecure information flow in existing Web applications. The empirical study performed by the authors indicates that steps must be taken to mitigate the privacy threat from covert flows in browsers.

4.4. Libraries

In addition to external information-flow analyses, i.e., analyses implemented outside of the target programming langue, a line of work strives towards achieving similar guarantees by exploiting existing programming language features [95,90,90]. This has the advantage that the existing programming language infrastructure can be used without any modifications.

Conti and Russo [95] show how to provide a taint mode via a library in Python. The library is able to keep track of tainted values for several built-in classes and supports propagation of taint information. The library uses decorators as a noninvasive approach to mark source code with no or minimal modification in the code. A line of work investigates information flow libraries for Haskell. Li and Zdancewic [59] present a library for secure information flow in Haskell, providing a starting point for this line of

work. Russo et al. [90] provide a library for secure information flow in Haskell using monads. The library provides combinators for declassification related to the who, when and what dimensions. Tsai, Hughes and Russo[110] present an extension to [59] adding side-effectful computations and threads. DelTedesco et al. [36] implement information erasure as a Python library building on ideas for dynamic taint analysis.

5. Conclusion

We have given an overview of the state-of-the-art in confidentiality and integrity policies and their enforcement. Our presentation of the confidentiality and integrity policies is based on a systematic formalization of four dominant formulations of noninterference: termination-insensitive, termination-sensitive, progress-insensitive, and progress-sensitive, cast in the setting of two minimal while languages. In the account of information-flow enforcement, we have discussed highlights from static, dynamic, and hybrid program analysis for security, as well as security enforcement by libraries.

Acknowledgments

This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR.

References

[1] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

[2] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. Technical report, George Mason University, 2004.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, October 2008.

[4] A. Askarov and A. C. Myers. A semantic framework for declassification and endorsement. In *Proc. European Symp. on Programming*, LNCS, March 2010.

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

[6] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 53–60, June 2007.

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

[8] Aslan Askarov and Andrei Sabelfeld. Catch me if you can: permissive yet secure error handling. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 45–57, 2009.

[9] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[10] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.

[11] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, May 2008.

[12] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, June 2002.

[13] Anindya Banerjee, Roberto Giacobazzi, and Isabella Mastroeni. What you lose is what you leak: Information leakage in declassification policies. In *Proceedings of the Twenty-Third Conference on Mathematical Foundations of Programming Semantics (MFPS)*, 2007.

[14] J. Barnes and JG Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[15] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 17–30, Berkeley, CA, USA, 2008. USENIX Association.

[16] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. European Symp. on Research in Computer Security*, volume 4734 of *LNCS*, pages 2–18. Springer-Verlag, September 2007.

[17] G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In *Proc. FLOPS*, volume 1722 of *LNCS*, pages 53–67. Springer-Verlag, November 1999.

[18] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[19] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *Proceedings of the International Conference on Information Systems Security (ICISS)*. Springer-Verlag, 2010.

[20] Aaron Bohannon and Benjamin C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In *Usenix Conference on Web Application Development (WebApps)*, pages 123–134, 2010.

[21] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, November 2009.

[22] N. Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *LNCS*, 2006.

[23] Niklas Broberg and David Sands. Flow-sensitive semantics for dynamic information flow policies. In S. Chong and D. Naumann, editors, *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS 2009)*, Dublin, June 15 2009. ACM.

[24] Niklas Broberg and David Sands. Paralocks – role-based information flow control and beyond. In *POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

[25] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.

[26] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

[27] S. Chong and A. C. Myers. Language-based information erasure. In *Proc. IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005.

[28] S. Chong and A.C. Myers. Decentralized robustness. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, 2006.

[29] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[30] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. *SIGPLAN Not.*, 44:50–62, June 2009.

[31] D. Clark and S. Hunt. Noninterference for deterministic interactive programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, October 2008.

[32] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation, Special Issue on Lambda-calculus, type theory and natural language*, 18(2):181–199, 2005.

[33] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[34] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[35] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K.

Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[36] Filippo Del Tedesco, Alejandro Russo, and David Sands. Implementing erasure policies using taint analysis. In Tuomas Aura, editor, *The 15th Nordic Conference in Secure IT Systems*, LNCS. Springer Verlag, October 2010.

[37] D. Demange and David Sands. All Secrets Great and Small. In *Programming Languages and Systems. 18th European Symposium on Programming, ESOP 2009*, number 5502 in LNCS, pages 207–221. Springer Verlag, 2009.

[38] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

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

[40] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[41] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 1–17, June 2002.

[42] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

[43] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[44] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 186–197, January 2004.

[45] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 295–310. Springer-Verlag, April 2005.

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

[47] J. Guttman. Invited tutorial: Integrity. Presentation at the Dagstuhl Seminar on Mobility, Ubiquity and Security, February 2007. `http://www.dagstuhl.de/07091/`.

[48] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.

[49] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 79–90, 2006.

[50] S. Hunt and David Sands. Just forget it – the semantics and enforcement of information erasure. In *Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008*, number 4960 in LNCS, pages 239–253, 2008.

[51] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 270–283, New York, NY, USA, 2010. ACM.

[52] J. Landauer and T. Redmond. A lattice of information. In *Proc. IEEE Computer Security Foundations Workshop*, pages 65–70, June 1993.

[53] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, July 2007.

[54] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006. To appear.

[55] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Workshop on Formal Aspects in Security and Trust (FAST'03)*, 2003.

[56] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, January 2005.

[57] P. Li and S. Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Workshop on Foundations of Computer Security*, pages 45–54, June 2005.

[58] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. *SIGPLAN Not*, pages 158–170, 2005.

[59] Peng Li and Steve Zdancewic. Encoding information flow in haskell. *Computer Security Foundations*

Workshop, IEEE, 0:16, 2006.

[60] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.

[61] G. Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.

[62] A. Lux and H. Mantel. Who can declassify? In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, volume 5491 of *LNCS*, pages 35–49. Springer-Verlag, 2009.

[63] S. Maffeis, J.C. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proc of ESORICS'09*. LNCS, 2009.

[64] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, April 2010.

[65] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *15th Nordic Conference on Secure IT Systems*, 2010.

[66] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proceedings of the IFIP International Information Security Conference (SEC)*, September 2010.

[67] H. Mantel. Information flow control and applications—Bridging a gap. In *Proc. Formal Methods Europe*, volume 2021 of *LNCS*, pages 153–172. Springer-Verlag, March 2001.

[68] H. Mantel. On the composition of secure systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 81–94, May 2002.

[69] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, March 2007.

[70] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 126–142, June 2001.

[71] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Computer Security*, 11(4):615–676, September 2003.

[72] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004.

[73] Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. *Computer Security Foundations Workshop, IEEE*, 0:226–240, 2005.

[74] Ana Almeida Matos, Gérard Boudol, and Ilaria Castellani. Typing noninterference for reactive programs. *Journal of Logic and Algebraic Programming*, 72(2):124 – 156, 2007. Programming Language Interference and Dependence.

[75] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.

[76] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[77] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[78] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.

[79] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2006.

[80] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[81] K. O'Neill, M. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201, July 2006.

[82] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *J. Functional Programming*, 7(6):557–591, 1997.

[83] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing (4th Edition)*. Prentice Hall, 2006.

[84] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, New York, NY, USA, 2009. ACM.

[85] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and*

Privacy, pages 102–113, May 1995.

[86] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, January 2003.

[87] A. W. Roscoe. CSP and determinism in security modeling. In *Proc. IEEE Symp. on Security and Privacy*, pages 114–127, May 1995.

[88] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. IEEE Computer Security Foundations Workshop*, pages 228–238, June 1999.

[89] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

[90] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24. ACM, 2008.

[91] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

[92] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis, April 2009. Draft.

[93] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[94] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, September 2009.

[95] Alejandro Russo and Juan José Conti. A taint mode for python via a library. In *OWASP AppSec Research*, 2010.

[96] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[97] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, September 2002.

[98] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[99] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

[100] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[101] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.

[102] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[103] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, January 2009.

[104] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 2000. IEEE Computer Society Press.

[105] Ravi S. Sandhu. On five definitions of data integrity, 1993.

[106] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.

[107] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[108] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.

[109] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis for Java. In *ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, June 2009.

[110] Ta Chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, July 2007.

[111] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention

with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, February 2007.

[112] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, September 1999.

[113] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[114] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, November 1999.

[115] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, January 2000.

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

[117] WebSand. Deliverable 1.1: Consolidation of state-of-the-art. January 2011.

[118] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

[119] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 46–61. Springer-Verlag, April 2001.