Decentralized Delimited Release

Jonas Magazinius[1], Aslan Askarov[2], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology
[2] Cornell University

Abstract. Decentralization is a major challenge for secure computing. In a decentralized setting, principals are free to distrust each other. The key challenge is to provide support for expressing and enforcing expressive decentralized policies. This paper focuses on *declassification* policies, i.e., policies for intended information release. We propose a decentralized language-independent framework for expressing what information can be released. The framework enables combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it. We instantiate the framework for a simple imperative language to show how the decentralized declassification policies can be enforced by a runtime monitor and discuss a prototype that secures programs by inlining the monitor in the code.

1 Introduction

Decentralization is a major challenge for secure computing. In a decentralized setting, principals are free to distrust each other. The key challenge is to provide support for expressing and enforcing expressive decentralized policies. Decentralization is of major concern for language-based information-flow security [42]. Information-flow security ensures that the flow of data through program constructs is secure. Information-flow based techniques are helpful for establishing end-to-end security. For example, a common security goal is *noninterference* [16, 21, 48, 42] that demands that public output does not depend on secret input. There has been much progress on tracking information flow in languages of increasing complexity [42], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [36, 47, 38].

A particularly important problem in the context of information-flow security is *declassification* [46] policies, i.e., policies for intended information release. These policies are intended to allow some information release as long as the information release mechanisms are not abused to reveal information that is not intended for release. Revealing the result of a password check is an example of intended information release, while revealing the actual password is unintended release. Similarly, the average grade for an exam is an example of intended information release, while revealing the individual grades of all students is unintended release. Abusing the underlying declassification mechanism for unintended release constitutes *information laundering*.

Decentralization makes declassification particularly intriguing. When is a piece of data allowed to be released? The answer might be simple when the piece of data originates from a single principal and needs to be passed to another one. However, when the piece of data originates from several sources, data release needs to satisfy security

requirements of all parties involved. Despite a large body of work on declassification (discussed in Section 5), providing a clean semantic treatment for decentralized declassification has been so far out of reach. Concretely, the unresolved challenge we address is prevention of information laundering in decentralized security policies.

Consider a scenario of a web mashup. A *web mashup* is a web service that integrates a number of independent services into a single web service. A common example is a mashup that combines information on available apartments and a map service (such as Google Maps) in an interactive service that displays apartments for sale on a map. Components of a mashup typically originate from different Internet domains.

A crucial challenge when building secure mashups [17] is hitting the sweet spot between separation and integration. The components need to communicate with each other but without stealing sensitive information. For example, a mashup that displays trucks with dangerous goods on a map might reveal the corner points of a required map to the map service but it must not reveal sensitive information about displayed objects such as the type of dangerous goods [26].

Collaboration in the presence of mutual distrust requires solid policy and enforcement support. Pushing the mashup scenario further, consider two web services (say, Gmail and Facebook) that are willing to swap sensitive information under the condition that both provide their share. For example, this might be a client-side mashup that allows cross-importing Gmail's and Facebook's address books. We want the policy framework to support the swap but prevent stealing Gmail's address book by Facebook.

A prominent line of work on declassification in a decentralized setting is the *decentralized label model (DLM)* [32]. This model underlies the security labels tracked by the Java-based information-flow tracker Jif [36]. DLM labels explicitly records owners. Owners are allowed to introduce arbitrary declassification on the part of labels they own. However, no soundness arguments for Jif's treatment of the labels are provided.

While inspired by DLM, our goal is precise semantic specification of decentralized security and its sound enforcement. Our focus is on exactly what can be released, which prevents information laundering. Unlike the DLM enforcement as performed by Jif, we do distinguish between programs that reveal the result of matching against a password from programs that reveal the password itself.

Combining the decentralization in the fashion of DLM and the laundering prevention in the fashion of *delimited release* [43], this paper proposes a decentralized language-independent framework for expressing what information can be released. The framework enables release of combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it.

To illustrate that the framework is realizable at language level, we instantiate the framework for a simple imperative language to show how the decentralized declassification policies can be enforced by a runtime monitor. We resolve the challenge of respecting decentralized policies while at the same time preventing laundering. Further, the monitor allows on-the-fly addition of new declassification polices by different principles. The monitor provides a safe approximation for the security policy. As it is often the case with automatic enforcement of nontrivial policies, the monitor is incomplete in the sense that some secure runs are blocked.

Further, we have implemented a prototype for a small subset of JavaScript that secures programs by inlining the information-flow monitor in the code. The inlining transformation transforms an arbitrary, possibly insecure, program into one that performs inlined information-flow checks, so that the result of the transformation is secure by construction.

2 Decentralized delimited release

Principals and security levels Our model is built upon a notion of *decentralized principals* which we denote via p, q. We assume that principals are mutually distrusting and that there are no "actsfor" or "speaks-for" relations [33, 23] between them.

We consider a *lattice of security levels* $\mathcal{L}$ and denote by $\sqsubseteq$ the ordering between elements of the lattice. A simple security lattice consists of two elements L and H, such that $L \sqsubseteq H$ i.e., L is no more restrictive than H. The structure of the security lattice does not have to be connected to principals in general, though they may be related as illustrated in Section 2.2.

We assume that different parts of global state (or memory) are labeled with different security levels: the higher the security level, the more sensitive the information which is labeled with that level. We also associate every security level in our model with an adversary that may observe memory states at that level: the higher the security level, the more powerful the adversary associated with that level. For two-level security lattice, an adversary corresponding to level L can observe only *low* (or public) parts of the state, while adversary corresponding to level H can observe all parts of the state.

Policies as equivalence relations Our model uses *partial equivalence relations* (PERs) over memories for use in confidentiality policies [1, 45]. The PER representation allows for fine granularity in individual policies. We believe that intentional models of security such as DLM [33] or tag-based models [19, 22, 49, 12] can be easily interpreted using PERs. Section 2.2 is an example of one such translation for a simple subset of DLM.

Intuitively, two memories m and m' are indistinguishable according to an equivalence relation I if $m\ I\ m'$. Two particular relations that we use are Id and All introduced by the following definition:

Definition 1 (Id and All relations) *Assuming that $\mathcal{M}$ ranges over all possible memories, define $Id \triangleq \{(m, m) \mid m \in \mathcal{M}\}$ and $All \triangleq \{(m, m') \mid m, m' \in \mathcal{M}\}$*

Assume an extension of memory mappings from variables to expressions, so that $m(e)$ corresponds to the value of expression e in memory m. We also introduce an indistinguishability induced by a particular set of expressions.

Definition 2 (Indistinguishability induced by $\mathcal{E}$) *Given a set of expressions $\mathcal{E}$, define* indistinguishability induced by $\mathcal{E}$ *as* $\mathsf{Ind}(\mathcal{E}) \triangleq \{(m, m') \mid \forall e \in \mathcal{E}\ .\ m(e) = m'(e)\}$.

In set-theoretical terminology, operator $\mathsf{Ind}(\mathcal{E})$ is the *kernel* of the function that maps memories to values according to a given expression. When $\mathcal{E}$ consists of a single expression e we often write $\mathsf{Ind}(e)$ instead of $\mathsf{Ind}(\{e\})$.

Restriction We define an operator of *restriction* induced by a set of variables. The operator is handy in the following examples and in the translation in Section 2.1.

Definition 3 (Restriction induced by variables X) *Given a set of variables X, define* restriction induced by X *to be a relation* $\mathsf{S}(X) \triangleq \mathsf{Ind}(\{y \mid y \notin X\})$ *i.e., indistinguishability relation for all variables y that are different from the ones in X.*

It can be easily shown that for disjoint sets of variables X and Y it holds that $\mathsf{S}(X \cup Y) = \mathsf{S}(X) \cup \mathsf{S}(Y)$. We often omit the set notation and write $\mathsf{S}(x, y)$ for $\mathsf{S}(\{x, y\})$.

Example: Consider memory with three variables x, y and z, and relation $\mathsf{S}(z)$. According to Def. 3, $\mathsf{S}(z) = \mathsf{Ind}(\{x, y\}) = \{m, m' \mid m(x) = m'(x) \wedge m(y) = m'(y)\}$. Here $\mathsf{S}(z)$ relates memories that must agree on all variables but z. In particular, given memories m_1 in which $x \mapsto 1, y \mapsto 1, z \mapsto 1$, m_2 in which $x \mapsto 1, y \mapsto 1, z \mapsto 0$, and m_3 in which $x \mapsto 1, y \mapsto 2, z \mapsto 1$ we have that $m_1\ \mathsf{S}(z)\ m_2$ but not $m_1\ \mathsf{S}(z)\ m_3$.

Confidentiality policies *Confidentiality policy* is a mapping from security levels in $\mathcal{L}$ to corresponding indistinguishability relations. Consider an example security lattice consisting of three security levels L, M, H, such that $L \sqsubseteq M \sqsubseteq H$. Assume also that our memory contains two variables x and y, and consider a confidentiality policy I such that $I(L) = All, I(M) = \mathsf{S}(x)$, and $I(H) = Id$

According to this policy, an attacker at level L can observe no part of the state, which is specified by $I(L) = All$. An attacker at level M can not observe the value of x but may observe the value of y. This is specified by using a restriction induced by x for $I(M)$. Finally, $I(H)$ establishes that an attacker at level H can observe all variables.

Say that a confidentiality policy I is *well-formed* when $I(\top) = Id$, where $\top$ is the most restrictive element in $\mathcal{L}$. Moreover, for any two labels $\ell \sqsubseteq \ell'$ it must hold that $I(\ell) \supseteq I(\ell')$. Our example policy above is well-formed. Indeed, $I(H) = Id$ and $I(L) = All \supseteq I(M) = \mathsf{S}(x) = \mathsf{Ind}(y) \supseteq I(H) = Id$. It is also easy to show that a policy obtained from point-wise union and intersection of well-formed policies is well-formed. The rest of the paper assumes that all policies are well-formed.

2.1 Decentralized policies

In a decentralized setting every principal provides its confidentiality policy. We denote a confidentiality policy of principal p as I_p. In particular, $I_p(\ell)$ is a relation specifying what memories must be indistinguishable at levels ℓ and below according to principal p. Given two principals p and q with policies I_p and I_q, the combination of these policies is policy I' s.t. for all ℓ we have $I'(\ell) = I_p(\ell) \cup I_q(\ell)$. Note that I' combines restrictions of both p and q and is as restrictive as both I_p and I_q. The following definition generalizes combination of trusted policies.

Definition 4 (Combination of confidentiality policies) *Given a number of principals $p_1 \ldots p_n$ with policies I_{p_i}, $1 \leq i \leq n$, the combination of these policies is a policy I' such that for all ℓ it holds that $I'(\ell) = \bigcup_i I_{p_i}(\ell)$.*

Example: Consider a lattice with three levels L, M, and H as before and a memory with two variables x and y. Consider two principals p and q with the policies $I_p(L) = All, I_p(M) = \mathsf{Ind}(x)$, $I_p(H) = Id$, $I_q(L) = All, I_q(M) = \mathsf{Ind}(y), I_q(H) = Id$. According to these policies p and q have different views on what can be observable

at level M. Combining these two policies, we obtain a policy I', such that $I'(L) = All, I'(M) = All$, and $I'(H) = Id$. Combining restrictions of both p and q means $I'(M)$ allows an attacker at level M to observe neither x nor y.

Declassification Declassification corresponds to relaxing individual policies I_p. We assume that every principal provides a set of *escape hatches* [43] that correspond to that principal's view on what data can be declassified.

Definition 5 (Escape hatches) *An escape hatch is a pair (e, ℓ) where e is a declassification expression, and ℓ is a level to which e may be declassified.*

Given a set of escape hatches $\mathcal{E}_p$ for principal p and an initial indistinguishability policy of this principal I_p we can obtain a less restrictive indistinguishability policy as follows.

Definition 6 *Given a confidentiality policy I and a set of escape hatches $\mathcal{E}$, let* declassification operator D *return a policy that relaxes I with $\mathcal{E}$. We define D pointwise for every level ℓ so that $\mathsf{D}(I, \mathcal{E})(\ell) \triangleq I(\ell) \cap \mathsf{Ind}(\mathcal{E}_\ell)$ where $\mathcal{E}_\ell = \{e \mid (e, \ell') \in \mathcal{E} \wedge \ell' \sqsubseteq \ell\}$ is the selection of escape hatches from $\mathcal{E}$ that are observable at ℓ.*

Example: Consider I_p as in Section 2.1 and escape hatch (y, L). Let us assume $I' = \mathsf{D}(I_p, \{(y, L)\})$. We have $I'(L) = \mathsf{Ind}(x), I'(M) = Id$, and $I'(H) = Id$.

Example: declassification and composite policies. Consider again memory with two variables x and y, a simple two-level security lattice with security levels L and H such that $L \sqsubseteq H$, and two principals p and q. Assume that p's policy specifies that a low attacker cannot observe x, and that q specifies that low observer cannot observe any parts of the memory. The corresponding security policies can be given by the second and third columns of Figure 1, where $\mathsf{S}(x) = \mathsf{Ind}(y)$. The combination of policies of both p and q at level L is $\mathsf{Ind}(y) \cup All = All$. That is, principals agree on no information being observable to an adversary at the level L.

ℓ	$I_p(\ell)$	$I_q(\ell)$	$\mathsf{D}(I_p, \mathcal{E}_p)(\ell)$	$\mathsf{D}(I_q, \mathcal{E}_q)(\ell)$
H	Id	Id	Id	Id
L	$\mathsf{S}(x)$	All	$\mathsf{S}(x) \cap \mathsf{Ind}(x)$	$\mathsf{Ind}(x+y)$

Fig. 1: Declassification and composite policies

Assume principal p declassifies the value of x to L, and principal q declassifies the value of $x + y$ to L, i.e., $\mathcal{E}_p = \{(x, L)\}$ and $\mathcal{E}_q = \{(x + y, L)\}$. The corresponding policies are given by the last two columns of Figure 1. The result of combining policies at level L is captured by the relation $(\mathsf{Ind}(y) \cap \mathsf{Ind}(x)) \cup \mathsf{Ind}(x+y)$ which is equivalent to $\mathsf{Ind}(x + y)$. That is, both principals allow $x + y$ to be observed at level L.

Security Our security condition is based on decentralized confidentiality policies. For generality, this section uses an abstract notion of a *system with memory*, denoted by $S(\cdot)$. A transition of system $S(m)$ with memory m to a *final* state with memory m' is written as $S(m) \Downarrow m'$. Section 3 instantiates this abstraction with standard program configurations. We call our security condition *decentralized delimited release* (DDR).

Definition 7 (Batch-style DDR) *Assume principals $p_1, \ldots, p_n$ with confidentiality policies $I_1 \ldots I_n$ and declassification policies given by escape hatch sets $\mathcal{E}_1 \ldots \mathcal{E}_n$. Say that a system with memory $S(\cdot)$ satisfies* decentralized delimited release *when for every level ℓ and for all memories m_1, m_2 for which $m_1 \bigcup_{1 \le i \le n} \mathsf{D}(I_i, \mathcal{E}_i)(\ell)\ m_2$ it holds that whenever $S(m_1) \Downarrow m_1'$ and $S(m_2) \Downarrow m_2'$ it must be that $m_1' \bigcup_i I_i(\ell)\ m_2'$.*

DDR borrows its intuition from the original definition of delimited release [43], and generalizes it to the case of several principles. In fact, in case of a single principal this definition matches the original definition in [43].

The key element of this definition is that it prevents *laundering* attacks. To see an example of a laundering attack, consider the following examples. Assume a memory with three variables x, y, z and individual policies of two principals p and q, as shown in the second and third columns of Figure 2. Here $\mathsf{S}(x, y)$ is restriction induced by x and y, and $\mathsf{S}(x, y) = \mathsf{Ind}(z)$, i.e., this relation allows only variable z to be observable.

Assume escape hatch sets where p declassifies $x + y$ to L, i.e., $\mathcal{E}_p = \{(x + y, L)\}$, and q declassifies both x and y individually to L, i.e., $\mathcal{E}_q = \{(x, L), (y, L)\}$. Taking the escape hatches into account we obtain the relations shown by the last two columns of Figure 2. According to these policies the program $z := x+y$ is secure. On the other hand the program $x := y; z := x + y$ is insecure. To see this consider two memories m_1 and m_2 where in m_1 we have $x \mapsto 1, y \mapsto 1, z \mapsto 0$ and in m_2 we have $x \mapsto 0, y \mapsto 2, z \mapsto 0$. We have that $m_1\ \mathsf{D}(I_p, \mathcal{E}_p)(L) \cup \mathsf{D}(I_q, \mathcal{E}_q)(L)\ m_2$, but not $m'_1\ I_p(L) \cup I_q(L)\ m'_2$.

DLM^0 We adopt the Decentralized Label Model (DLM) [32] as our model of expressing security policies sans actsfor relation, that we dub DLM^0 . We nevertheless, retain top and bottom principals $\bot$ and $\top$ that allow us to express the most and the least restrictive security policies. In DLM a security level of a variable records *policy owners*, reviewed below. On the intuitive level policy owner is a principal who cares about the sensitivity of the data. This is more than simply a principal who can read data — not every principal who reads data is necessarily interested in preserving its confidentiality.

DLM policies are the basic building blocks for expressing security restrictions by principals. A (confidentiality) policy is written $o \to r_1, \ldots, r_n$, where o is the owner of the policy, and $r_1, \ldots, r_n$ is the set of readers. Here principal o restricts the flow of data to the principals in the readers set. For example, in the policy $Alice \to Bob, Carol$ Alice constraints the set of readers to only Bob, Carol, and herself (the owner is implicitly a reader). Similarly, a policy $Carol \to Carol$ restricts all but Carol from reading data.

Security labels, denoted by ℓ, are either DLM policies or are composed from other labels in one of the two ways: (i) conjunction of two labels, written $\ell_1 \sqcup \ell_2$, is a label that enforces restrictions of both ℓ_1 and ℓ_2. (ii) disjunction of two labels, written $\ell_1 \sqcap \ell_2$, is a label that enforces restrictions of either ℓ_1 or ℓ_2. An example of a conjunction label is $\{Alice \to Bob, Carol\} \sqcup \{Carol \to Carol\}$. Carol is the only reader; because of the Carol's policy, this label restricts either Alice or Bob from reading data. Disjunction label $\{Alice \to Alice\} \sqcap \{Bob \to Bob\}$ allows both Alice and Bob to read data.

Labels can be ordered by the "no more restrictive than" [34, 14] relation: $\ell_1 \sqsubseteq \ell_2$ when ℓ_1 restricts data no more than ℓ_2 does. We use $\{\bot \to \bot\}$ to denote the least restrictive label (also denoted simply $\bot$), i.e., for all ℓ it holds that $\{\bot \to \bot\} \sqsubseteq \ell$. For example, $\{Alice \to Alice, Bob\} \sqsubseteq \{Alice \to Alice\}$, because in the right label, Alice

ℓ	$I_p(\ell)$	$I_q(\ell)$	$\mathsf{D}(I_p, \mathcal{E}_p)(\ell)$	$\mathsf{D}(I_q, \mathcal{E}_q)(\ell)$
H	Id	Id	Id	Id
L	$\mathsf{S}(x, y)$	$\mathsf{S}(x, y)$	$\mathsf{S}(x, y) \cap \mathsf{Ind}(x + y)$	$\mathsf{S}(x, y) \cap \mathsf{Ind}(x) \cap \mathsf{Ind}(y)$

Fig. 2: Policies for example laundering attack

imposes stricter restrictions by allowing only her to be the reader. However (assuming there is no actsfor relationship between Alice and Bob), $\{Alice \to Bob\} \not\sqsubseteq \{Bob \to Alice\}$. Here Alice's constraints are not satisfied. Her label on the left restricts the flow to Bob, but there are no Alice's policies on the right.

2.2 From DLM^0 to families of indistinguishability relations

This section shows how DLM^0 labels can be translated to confidentiality policies. The translation is parametrized by the principals. We define two operators in this translation — the top level translation operator $\widetilde{\mathsf{T}}_p$ and a helper operator T_p. The top level translation operator $\widetilde{\mathsf{T}}_p$, that returns a confidentiality policy for principal p, takes the variable environment Γ as a single argument. It is defined so that when $\Gamma = \varnothing$ then in the resulting confidentiality policy $\widetilde{\mathsf{T}}_p(\Gamma)$, the corresponding indistinguishability relation for all labels ℓ is Id. This indeed matches the DLM intuition that no restrictions imply the most permissive confidentiality policy. To translate restrictions that are captured by DLM labels, we define a helper operator $\mathsf{T}_p(I_p, \ell, x)$.

Definition 8 (Translation of a single label T_p) *Given a principal p with an initial policy I_p, label ℓ, and variable x, define $\mathsf{T}_p(I_p, \ell, x)$ inductively based on the structure of ℓ.*

case ℓ is an empty label *Return I_p.*
case ℓ is $\ell' \sqcup \{q \to \vec{r}\}$ such that $q \neq p$ and $q \neq \top$ *Return $\mathsf{T}_p(I_p, \ell', x)$.*
case ℓ is $\ell' \sqcup \{q \to \vec{r}\}$ such that $q = p$ or $q = \top$ *Define policy I'_p, where for all ℓ'' let*

$$I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup \mathsf{S}(x) & \text{if } \{q \to \vec{r}\} \not\sqsubseteq \ell'' \\ I_p(\ell'') & \text{otherwise} \end{cases} \text{ and return } \mathsf{T}_p(I'_p, \ell', x).$$

case ℓ is $\ell' \sqcap \{q \to \vec{r}\}$ such that $q = p$ or $q = \top$ *Define policy I'_p where for all ℓ'' let*

$$I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup \mathsf{S}(x) & \text{if } \{q \to \vec{r}\} \not\sqsubseteq \ell'' \wedge \ell' \not\sqsubseteq \ell'' \\ I_p(\ell'') & \text{otherwise} \end{cases} \text{ and return } \mathsf{T}_p(I'_p, \ell', x).$$

With the definition of T_p at hand we define the top-level translation operator $\widetilde{\mathsf{T}}_p$.

Definition 9 (Translation of DLM^0 policies) *Assume that Γ maps variables to DLM^0 labels. Define an operator $\widetilde{\mathsf{T}}_p$ that translates restrictions recorded in Γ to confidentiality policies as follows. We let $\widetilde{\mathsf{T}}_p(\varnothing) = \widetilde{Id}$, when $\Gamma = \varnothing$, and otherwise $\widetilde{\mathsf{T}}_p(x \mapsto \ell; \Gamma') = \mathsf{T}_p(\widetilde{\mathsf{T}}_p(\Gamma'), \ell, x)$. Here $\widetilde{Id}$ is a policy s.t. for all levels ℓ it holds $\widetilde{Id}(\ell) = Id$.*

Example: Consider memory consisting of four variables x, y, z and w. Assume two principals p and q, and variable environment Γ, s.t. $\Gamma(x) = \{p \to p\}, \Gamma(y) = \{q \to q\}, \Gamma(z) = \{p \to p, q\} \sqcup \{q \to p, q\}$, and $\Gamma(w) = \{p \to p\} \sqcap \{q \to q\}$. Translation of labels in Γ is represented by the second and third columns in the table below.

ℓ	$\widetilde{T}_p(\Gamma)(\ell)$	$\widetilde{T}_q(\Gamma)(\ell)$	$\mathsf{D}(\widetilde{T}_p(\Gamma), \mathcal{E}_p)(\ell)$	$\mathsf{D}(\widetilde{T}_q(\Gamma), \mathcal{E}_q)(\ell)$
$\{\top \to \top\}$	Id	Id	Id	Id
$\{p \to p\}$	Id	$\mathsf{S}(y)$	Id	$\mathsf{S}(y) \cap \mathsf{Ind}(x + y)$
$\{q \to q\}$	$\mathsf{S}(x)$	Id	$\mathsf{S}(x) \cap \mathsf{Ind}(x + y)$	Id
$\{p \to p, q\} \sqcup \{q \to p, q\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$	$\mathsf{S}(x) \cap \mathsf{Ind}(x + y)$	$\mathsf{S}(y) \cap \mathsf{Ind}(x + y)$
$\{p \to p\} \sqcap \{q \to q\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$	$\mathsf{S}(x)$	$\mathsf{S}(y)$
$\{\bot \to \bot\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$	$\mathsf{S}(x)$	$\mathsf{S}(y)$

Here $\mathsf{S}(x) = \mathsf{Ind}(y) \cap \mathsf{Ind}(z) \cap \mathsf{Ind}(w)$ and $\mathsf{S}(y) = \mathsf{Ind}(x) \cap \mathsf{Ind}(z) \cap \mathsf{Ind}(w)$. Consider escape hatches provided by each principals such that $\mathcal{E}_p = \mathcal{E}_q = \{(x + y, \{p \to p, q\} \sqcup \{q \to p, q\})\}$. Taking escape hatches into account the policies obtained from declassification operator are illustrated in fourth and fifth columns of the table above.

3 Enforcement

This section illustrates the realizability of our framework for a simple imperative language. We formalize the language along with a runtime enforcement mechanism that ensures security.

Language and semantics The syntax of the language is displayed in Figure 3. Expressions e operate on values n and variables x and might involve composition with operator op. Commands c are standard imperative commands. The only nonstandard primitive in the language is a declassification primitive $\texttt{declassify}(p, e, \ell)$ that declares an escape hatch (e, ℓ) of principal p.

$$e ::= n \mid x \mid e \ op \ e$$
$$c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{declassify}(p, e, \ell) \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c$$

Fig. 3: Syntax

$$\langle \texttt{declassify}(p, e, \ell), m\rangle \xrightarrow{d(p,e,\ell)} \langle stop, m\rangle \qquad \frac{m(e) = v}{\langle x := e, m\rangle \xrightarrow{a(x,e,m)} \langle stop, m[x \mapsto v]\rangle}$$

$$\frac{m(e) = n \qquad n \neq 0 \implies i = 1 \qquad n = 0 \implies i = 2}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \xrightarrow{b(e)} \langle c_i; end, m\rangle} \qquad \langle end, m\rangle \xrightarrow{f} \langle stop, m\rangle$$

Fig. 4: Monitored semantics: selected rules

$$\langle st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{b(e)} \langle lev_\Gamma(e) : st, i, \mathcal{E}, \Gamma\rangle \qquad \langle hd : st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{f} \langle st, i, \mathcal{E}, \Gamma\rangle$$

$$\langle st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{d(p,e,\ell)} \langle st, i, \mathcal{E}[p \mapsto \mathcal{E}_p \cup \{(e, \ell, lev(st))\}], \Gamma\rangle$$

$$\frac{lev(st) \sqsubseteq \Gamma(x) \qquad \ell \triangleq \mathsf{substEH}(lev_\Gamma(e), x, e, \mathcal{E}, \Gamma) \qquad lev_\Gamma(e) \not\sqsubseteq \ell \implies m(e) = i(e)}{\langle st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{a(x,e,m)} \langle st, i, \mathcal{E}, \Gamma[x \mapsto lev(st) \sqcup \ell]\rangle}$$

$$\begin{aligned}
&\mathsf{substEH}(\{\mathsf{o} \to \mathsf{r}\}, \mathsf{x}, \mathsf{e}, \mathcal{E}, \Gamma) \triangleq \{o \to \widetilde{r}\} \sqcap \{\ell \mid (e, \ell, pc) \in \mathcal{E}_o \wedge pc \sqsubseteq \Gamma(x)\}\\
&\mathsf{substEH}(\ell_1 \sqcup \ell_2, x, e, \mathcal{E}, \Gamma) \triangleq \mathsf{substEH}(\ell_1, x, e, \mathcal{E}, \Gamma) \sqcup \mathsf{substEH}(\ell_2, x, e, \mathcal{E}, \Gamma)\\
&\mathsf{substEH}(\ell_1 \sqcap \ell_2, x, e, \mathcal{E}, \Gamma) \triangleq \mathsf{substEH}(\ell_1, x, e, \mathcal{E}, \Gamma) \sqcap \mathsf{substEH}(\ell_2, x, e, \mathcal{E}, \Gamma)
\end{aligned}$$

Fig. 5: Monitor semantics: selected rules

Figure 4 contains the semantic rules for evaluating commands. A *memory* is a mapping from variables to values, where values range over some fixed set of values (say, without loss of generality, the set of integers). We assume an extension of memories to expressions that is computed using a semantic interpretation of constants as values and operators as total functions on values. This allows us writing $m(e)$ for the value of

expression e in memory m. A *configuration* has the form $\langle c, m\rangle$ where c is a command in the language and m is a memory. A *transition* has the form $\langle c, m\rangle \xrightarrow{\beta} \langle c', m'\rangle$ representing a computation step from configuration $\langle c, m\rangle$ to $\langle c', m'\rangle$. *Events* β are there to communicate relevant information about program execution to an execution monitor (this style of presenting monitors follows recent work on information-flow monitoring, e.g., [44, 5]). When events are unimportant, we may omit explicitly writing them out as in $\langle c, m\rangle \longrightarrow \langle c', m'\rangle$. The meaning of the particular events is spelled out in the description of the monitor below.

Monitor Our enforcement mechanism is a runtime monitor. Listening to a given program event, the monitor either grants execution (possibly updating its internal state) or blocks it. Following the idea sketched in [26], we obtain security by requiring two conditions on declassification (in addition to standard tracking "regular" flows orthogonal to declassification). The first condition is to check that all declassifications are allowed. The second condition ensures that the value of an escape hatch expression has not changed since the start of the program. The former is in charge of the *who* dimension of declassification, preventing release to unauthorized principals, whereas the latter controls the *what* dimension, preventing information laundering. Section 5 discuses these and other dimensions of declassification [46] in further detail.

Figure 5 presents selected monitor rules. Monitor configurations have the form $\langle st, i, \mathcal{E}, \Gamma\rangle$, where st is a stack of security levels, i stores the initial program memory, $\mathcal{E}$ is an indexed collection of sets of escape hatches, and Γ is the current security environment. Escape hatches are also extended to the form (e, ℓ, pc), where pc records the level of the monitor stack when that escape hatch has been added. The monitor features a form of flow-sensitivity: security level of a variable $\Gamma(x)$ can be updated, but only when the decision to update does not give away secret information [6].

Assume an overloaded function $lev(\cdot)$ that returns the least upper bound on the security level of components in the argument. For expressions, the components are the subexpressions and for lists the components are the list elements. When monitor stack is empty $lev(\cdot)$ is the least restrictive label $\bot \to \bot$.

The event $b(e)$ is generated by conditionals and loops when branching on an expression e. This is interesting information for the monitor because it introduces risks for *implicit information flow* [18] through control-flow structure of the program. For example, program `if` h `then` $l := 1$ `else` $l := 0$ leaks whether the initial value of (secret) variable h is (non)zero into the final value of (public) variable l. The essence of an implicit flow is a public side effect in a secret computation context. To record the computation context, we keep track of the security levels of the variables branched on. Thus, the monitor always accepts branching on an expression, pushing the level of the expression on the stack. The event f is generated by conditionals and loops on reaching a joint point of branching. The monitor always accepts this event, popping the top security level from the stack. The event $d(p, e, \ell)$ is generated upon declassification of expression e to level ℓ by principal p. In response, the monitor includes the newly declared escape hatch in its environment and records the current level of the stack $lev(st)$.

The event $a(x, e, m)$ is generated by assignment of an expression e to a variable x in memory m. First, the monitor blocks implicit flows by requiring that the level of the x is at least as restrictive as the least upper bound of the security levels on the stack. Next,

the monitor checks if this assignment can be treated as a declassification. The operator substEH performs a label substitution and returns the least restrictive label that can be obtained by using declassifications in $\mathcal{E}$. Note that all information used by substEH check is bounded by $\Gamma(x)$ — we only look up escape hatches that syntactically agree on expression e and that were updated in the contexts with $pc \sqsubseteq \Gamma(x)$. If expression can be declassified to a level that is more permissive than $lev(e)$, the monitor checks that the escape-hatch expression must be the same in the initial and current memories. This prevents information laundering as in $\texttt{declassify}(p, h, p \to \bot); h := h'; l := h$ where h is declared to be declassified whereas h' is actually leaked. Finally, the monitor updates the level of $\Gamma(x)$, featuring flow-sensitivity mentioned earlier in this Section.

The monitor accepts program $l := x + y$, if both A's and B's escape hatches contain $x + y$, and rejects it if either A or B do not explicitly list $x + y$ in their escape hatches.

While, as we will show, the enforcement is sound, it is obviously incomplete. In the setting of the example above, the program is rejected when A's escape-hatch set is $\{x\}$ and B's is $\{y\}$. A and B are willing to release all of their data, and so the program is rightfully accepted secure by the security definition. However, the monitor rejects the program because expression $x + y$ is not found in the escape-hatch sets.

Soundness The monitor guarantees secure execution in the presence of mutual distrust. We instantiate the notion of system with memories of Definition 7 with monitored program configurations $(\langle c, m\rangle, \langle st, i, \mathcal{E}, \Gamma\rangle)$. Assume all declassification policies are expressed in $\mathcal{E}$ and c contains no further `declassify` statements. This is consistent with our implementation (cf. Section 4) in which escape hatches are collected at parse time. We write $(\langle c, m\rangle, \langle st, i, \mathcal{E}, \Gamma\rangle) \Downarrow m', \Gamma'$ when $(\langle c, m\rangle, \langle st, i, \mathcal{E}, \Gamma\rangle) \longrightarrow^* (\langle stop, m'\rangle, \langle st', i, \mathcal{E}, \Gamma'\rangle)$, where $\longrightarrow^*$ is a transitive closure of $\longrightarrow$. Assume principals $p_1, \ldots, p_n$ with individual declassification policies $\mathcal{E}_{p_i}$. Formally, we have:

Theorem 1 (Soundness) *Assume principals $p_1, \ldots, p_n$ with initial DLM^0 policies expressed in the environment Γ and declassification policies expressed by the collection of sets of escape hatches $\mathcal{E}$, indexed by p_i. Consider program c free of* `declassify` *statements. Then for all levels ℓ and memories m_1, m_2 s.t. $m_1 \bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell)$ m_2 if $(\langle c, m_1\rangle, \langle \epsilon, m_1, \mathcal{E}, \Gamma\rangle) \Downarrow m_1', \Gamma_1'$, and $(\langle c, m_2\rangle, \langle \epsilon, m_2, \mathcal{E}, \Gamma\rangle) \Downarrow m_2', \Gamma_2'$, then $\bigcup_p \widetilde{\mathsf{T}}_p(\Gamma_1')(\ell) = \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma_2')(\ell)$ and $m_1' \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma_1')(\ell)$ m_2'.*

The proof of Theorem 1 is available in the accompanying technical report [27].

Example: We revisit the example with aggregate computation from Section 2.1. Consider variable environment consisting of three variables x, y and z. Assume two principals p and q s.t. $\Gamma(x) = \{p \to p\}, \Gamma(y) = \{q \to q\}$, and $\Gamma(z) = \{p \to p, q\} \sqcup \{q \to p, q\}$. and escape hatch sets for every principal s.t. $\mathcal{E}_p = \mathcal{E}_q = \{(x + y, \{p \to p, q\} \sqcup \{q \to p, q\}, \bot \to \bot)\}$. Then basic declassification of the form $z := x + y$ is accepted, while laundering as in the program $x := y; z := x + y$ is rejected.

4 Experiments

Next, we present the experiments conducted on enforcement of the monitor in practice. The inlining transformation converts a program in a language from Section 3 into a program in JavaScript with inlined security checks. In this experiment we have successfully implemented two scenarios in a restricted subset of JavaScript.

Experiment setup To implement runtime source transformation we need functionality for parsing and rewriting of JavaScript code, written in JavaScript. We use ANTLR [2] to generate such a parser/rewriter from a JavaScript grammar. The generated parser is 7650 LOC of JavaScript, not counting additional 165 LOC for the user-defined JavaScript and 6139 LOC in the runtime library. For performance, the code can be further reduced using JavaScript compression tools. All sources are available on demand.

The monitor must be inlined before the code is parsed by the browser, or else the code is executed unmonitored. The Opera browser [37] allows the user to include privileged JavaScript called "User JavaScript". User JavaScript can access functions and events not accessible to ordinary JavaScript, including the event "BeforeScript", that enables rewriting the script source before the source reaches the browser's parser. This allows us to inline the monitor whenever a new script is loaded.

This approach introduces two sources of runtime overhead. The first is the parsing and rewriting, performed once per code segment. The second is the execution of the inlined monitor. Previous work [29] shows the total overhead of 2–10 times the untransformed runtime, depending on the code structure of, the browser, and the system used.

One alternative to implementing the monitor is using aspect-oriented techniques along the lines of, e.g., [28]. However, such an implementation would demand low-level access to program operations. For example, performing an assignment or reaching a joint point must be observable events in order to serve as pointcuts.

Transformation The generated parser parses and, in the process, rewrites the code, transforming it on the fly. If the parser cannot parse the input it throws an error and the code is not evaluated by the browser. The monitored code is hence limited by the parser.

The source language is a subset of JavaScript, as described in Section 3. The target language is full JavaScript. This means there are no restrictions on the monitor itself, only on the code being monitored. We identify different stages in the transformation that are closely related to the stages of the browser as it requests content. While other JavaScript-specific features, such as prototyping and objects, would make an interesting complement, more research on how such features affect information-flow analysis is required before extending the language and incorporating them in the framework.

Transformation in stages Based on information available at a given moment, only certain actions can be taken. Thus, we distinguish between *parse-time* and *run-time*.

Parse-time As scripts are encountered we enumerate their origins and for each origin load the associated escape hatches and initial levels for variables. The scripts are parsed on the fly. During parsing, when a security critical part of the source is encountered, we rewrite the source inlining the monitor according to a set of rules. Because JavaScript lacks a declassification primitive, unlike the monitor in Section 3, escape hatches are defined at parse-time. Note that while it is clear at parse-time which variables are used in an expression, their run-time values are unknown. This is crucial for declassification as it relies on which variables are used in expressions to determine which information to declassify. This transformation is detailed below.

Run-time At run-time, as the program is evaluated, all variables have their actual values, but when following an execution path we lose information about

```
var x;   // User variable
var _x_; // Level of x
var __x; // Initial value of x
var _pc; // Special variable
```

Listing 1.1: Naming convention

the control-flow structure of the program. Thus, the inlining transformation needs to encode necessary control-flow structure information for the monitor. As the transformed script is executed, the monitor validates the inlined checks.

Shadow variables To track information flow in the program we use shadow variables. Two kinds of shadow variables are used: one for the level of the variable, and one for its initial value. The shadow variables that hold the initial values are set when the corresponding variable is declared, while the shadow variable that hold the level are updated whenever the corresponding variables are initially assigned. The set of shadow variables corresponds to Γ in the formal monitor. Also, a small set of monitor specific variables is described below.

To prevent the code being monitored from interfering with state of the monitor, the shadow variables must be isolated. One could create a separate namespace for shadow variables, with minimal impact on the source program. The drawback is mimicking the scoping and variable lookup mechanisms of JavaScript, to prevent clashes between equally named variables from different scopes. Implementing this can be non-trivial.

Antoher possibility is to reserve an infrequently used character, such as "_", for shadow variables, thereby excluding it from the set of allowed characters for identifiers in the source language. This would prevent valid code, according to the parser, from referring to variables using this character. The benefit in this case is that we can piggyback on JavaScripts built in scoping mechanism. The drawback is that we moderately restrict the set of valid programs. As a design choice, we chose this option. The chosen naming convention can be seen below in Figure 1.1.

Special variables A few special variables exist to store the state of the monitor at run-time. For tracking implicit informations flows, the level of the cusrrent execution context is stored in the special variable `_pc`. The `_pc` works like a stack and is updated whenever a new execution context is entered. The variable `_E` stores all escape hatches and their associated levels. Finally the variable `_init` stores all initial levels of variables as defined by the owner of each variable.

Transformation rules We focus on the interesting cases of the transformation: assignment, declassification, and branching.

Assignment and declassification Following the semantics in Figure 4, the transformed code updates both the value of variable being assigned and the level of the corresponding shadow variable. Which level it updates to depend on whether the assignment expression is in the set of escape hatch expressions or not. In the case of declassification, the level is determined from the escape hatch, otherwise the new level is determined from the variables used in the expression. When determining the level, the current level of the execution context (the `_pc`) is also considered.

```
// Implicit flow check
while(!_pc.leq(_x_));
if ('y+z' in _E) {
  // Laundering check
    while((__y+__z)!=(y+z));
    _x_=_pc.join(_E['y+z']);
} else {
    _x_=_pc.join(_y_,_z_)
}
x=y+z;
```

Listing 1.2: Assignment rule

Insecure upgrade refers to assignment of a lower level variable in a higher level context, implying an information leak [40]. Insecure upgrade is prevented by checking that the `_pc` is less than or equal to the level of the variable [6]. If it is not, the program gets

stuck. Information laundering through declassification is prevented by checking that the current value is the same as the initial value of the expression. If this check fails, the program gets stuck. Listing 1.2 gives an example of an assignment after transformation.

Branches To prevent implicit information flows, the monitor tracks the level of the context in each branch. When a branch is encountered, the current level of the `_pc` is stored. The `_pc` is updated with the join of its current level and the level of the expression that is branched upon. Each of the two alternative code paths are transformed and after the two branches join again, the level of the `_pc` before the branch is restored. In the implementation, management of the `_pc` is done through helper methods, e.g. `_pc.branch(_x_); if(x){...}; _pc.joinPoint();`.

Scenarios We have applied the transformation to two simple yet illustrative scenarios. We believe that the approach of using inline transformation and escape hatches for tracking information flow scales to more complex scenarios: no matter how complex the language is, the secure use of escape hatches is restricted to simple patterns (with no modification of data involved in them).

```
while(!_pc.leq(_d_));
if ('orderOf(f)/p' in _E) {
   while((orderOf(__f)/__p)!=
      (orderOf(f)/p));
   _d_=_pc.join(_E['orderOf(f)/p']);
} else
   _d_=_pc.join(_f_,_p_);
d=orderOf(f)/(10*p);
```

Listing 1.3: Scenario 1 transformed

```
while(!_pc.leq(_a_));
if ('a.concat(b)' in _E) {
   while((__a.concat(__b))!=
      (a.concat(b)));
   _a_=_pc.join(_E['a.concat(b)']);
} else
   _a_=_pc.join(_a_,_b_);
a = a.concat(b);
```

Listing 1.4: Scenario 2 transformed

Social E-commercing In this scenario we have an e-commerce site (A) and a social networking site (B) who have an agreement that the users of the social networking site get a discount (d) on the products of the e-commerce site if they recommend the store to their friends. The size of the discount is determined by the price (p) and the number of friends (f) that the user recommends the site to. To protect the privacy of the user, the social networking site does not want to release the exact number of friends so the discount is calculated by the following formula: $d = e(f,p) = \frac{orderOf(f)}{10*p}$. For declassification the A specifies the escape hatch $E(A) = \{(e(f,p), \bot)\}$. An example of the transformed code for this scenario is available in Listing 1.3. In this scenario A could maliciously try to find the exact number of friend recommendations, e.g. using either `var x=f;` or `while(x<f)x++;`. Regardless, since both explicit and implicit information-flows are tracked this information is labeled as belonging to B.

Contact Swap Consider a mashup where the user can synchronize his contact lists on several social networking sites. In this scenario we have a truly distributed and collaborative release of information. The sites need to collaborate on which contacts to share and whom to share them with. That is, the user might be unwilling to share the contacts marked as business associates across networks, but still want to share contacts marked as friends. A sample of the transformed scenario code is available in Listing 1.4. Here both A and B would need to declassify the expression `a.concat(b)` to the other. As can be seen in this sample, the rewritten code prevents potential attacks. Malicious code could try to launder some other information by assigning it to either `a` or `b`, as such `b=`

`secret; a=a.concat(b);`. However, the transformation of this code gets stuck in the initial value check since the value of b no longer matches its initial value.

5 Related work

There is a large body of work on declassification, much of which is discussed in Sabelfeld and Sands' recent overview [46]. The overview presents dimensions and principles of declassification. The identified dimensions correspond to *what* data is released, *where* and *when* in the program and by *whom*. The *what* and *where* dimensions and their combinations have been studied particularly intensively [30, 4, 8, 9, 5].

Our approach integrates the *what* and *who* dimensions. It is the *who* dimension that has received relatively little attention so far. The precursor to work on the *who* dimension in the language-based setting is the decentralized label model (DLM) [32]. DLM allows principals expressing ownership information as well as explicit read/write access lists in security labels. Chen and Chong [11] generalizes DLM to describe a range of owned policies from information flow and access control to software licensing.

Work on *robustness* [35, 3], addressed the *who* dimension by preventing attacker-controlled data from affecting *what* is released. Lux and Mantel [24] investigate a bisimulation-based condition that helps expressing who (or, more precisely, what input channels) may affect declassification.

Our approach builds on the *composite release* [26] policy that combines the *what* and *who* dimensions. The escape hatches express the *what* and the ownership of the principals of the escape-hatch policies expresses the *who*. However, for composite release to be allowed, the principals have to *syntactically* agree on escape hatches. This paper removes this limitation and generalizes the principal model to handle DLM. The experimental part is another added value with respect to the previous work [26].

Broberg and Sands [10] describe *paralocks*, a knowledge-based framework for expressing declassification and role-based access-control policies. Broberg and Sands show how to encode DLM's *actsFor* relation using paralocks. However, paralocks do not address the *what* dimension of declassification.

Our enforcement draws on the ideas sketched by us earlier [26], where we present considerations for practical enforcement of composite release. The formalization of the enforcement fits well into the modular framework [44, 5] for dynamic information-flow monitoring where the underlying program and monitor communicate through the interface of events. The *what* part of declassification is enforced similarly to [5], by ensuring that the values of escape-hatch expressions have not been modified. The paper extends the formalization of the enforcement with the *who* part.

Recent efforts approach inlining for information flow. Chudnov and Naumann [15] inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [40]. The monitor does not offer support for declassification. As in this work, Magazinius et al. [29] concentrate on inlining purely dynamic monitors under the no-sensitive-upgrade discipline. The distinct feature is inlining on the fly, which allows a smooth treatment of dynamic code evaluation. While the inlining rules [29] offer no support for declassification, it is still a useful starting point for our experiments in Section 4.

In the web setting, work on language-based sadboxing such as object capabilities (e.g., [31, 25]) is less related because separation does not allow information flow and intended release. The most closely related project is the Mozilla project FlowSafe [20]

that aims at extending Firefox with runtime information-flow tracking, where dynamic information-flow monitoring [6, 7] lies at its core.

6 Conclusion

We have presented a framework for specitying and enforcing decentralized information-flow policies. The policies express possibilities of collaboration in the environment of mutual distrust. By default, no information flow is allowed across different principals. Whenever principals are willing to collaborate, the policy framework ensures that a piece of data is revealed only if all owners of the data have provided sufficient authorization for the release. While the policy framework is independent, we have demonstrated that is realizable with language support. We have showed how to enforce security by runtime monitoring for a simple imperative language.

A major direction of future work is integrating support for decentralized security policies into the line of work on information-flow controls in a web setting, where we have already investigated the treatment of dynamic code evaluation [5], timeout events [39], and interaction with the DOM tree [41].

Another intriguing avenue for integration is with Chong's *required release* [13] policy. This policy ensures that if a principal promises to release a piece of data, then this piece of data must be released. Such a policy is an excellent fit for thwarting attempts of cheating. For example, suppose three principals have agreed on releasing the average of their three pieces of data to each other. However, a cheating principal might attempt to withdraw its escape hatch or declassify to a level that is not sufficient for the other principals to be able to access the result. These attempts can be prevented by required release, where principals must release data according to the declared policies.

Bibliography

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.

[2] ANTLR Parser Generator. http://www.antlr.org/.

[3] A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In *Proc. European Symp. on Programming*, LNCS. Springer-Verlag, 2010.

[4] 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.

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

[6] 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.

[7] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. Technical Report UCSC-SOE-09-34, University of California, Santa Cruz, 2009.

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

[9] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.

[10] N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *Proc. ACM Symp. on Principles of Programming Languages*, January 2010.

[11] H. Chen and S. Chong. Owned policies for information security. In *Proc. IEEE Computer Security Foundations Workshop*, June 2004.

[12] W. Cheng. *Information Flow for Secure Distributed Applications*. PhD thesis, Massachusetts Institute of Technology, September 2009.

[13] S. Chong. Required information release. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[14] S. Chong and A. C. Myers. Decentralized robustness. In *Proc. IEEE Computer Security Foundations Workshop*, pages 242–253, July 2006.

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

[16] 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.

[17] M. Decat, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Towards building secure web mashups. In *Proc. AppSec Research*, June 2010.

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

[19] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[20] B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, October 2009.

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

[22] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[23] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *Proc. ACM Symp. on Operating System Principles*, pages 165–182, October 1991. *Operating System Review*, 253(5).

[24] 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.

[25] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[26] 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.

[27] J. Magazinius, A. Askarov, and A. Sabelfeld. Decentralized delimited release. Technical report, Chalmers University of Technology, 2011. Available at `http://www.cse.chalmers.se/~d02pulse/ddr-tr.pdf`.

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

[29] 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.

[30] 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.

[31] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.

[32] 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.

[33] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

[34] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

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

[36] 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 2001–2009.

[37] Opera, User JavaScript. http://www.opera.com/docs/userjs/.

[38] Praxis High Integrity Systems. Sparkada examiner. Software release. `http://www.praxis-his.com/sparkada/`.

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

[40] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[41] 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.

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

[43] 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.

[44] 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.

[45] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proc. European Symp. on Programming*, volume 1576 of *LNCS*, pages 40–58. Springer-Verlag, March 1999.

[46] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, January 2009.

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

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

[49] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.