Server-driven Outbound Web-application Sandboxing
FP7-ICT-2009-5, Project No. 256964

https://www.websand.eu

Deliverable D3.1
Confidentiality and Integrity Policies

Abstract

This deliverable reports on the results from the WebSand work package on information-flow control. These results underpin a framework for decentralized confidentiality and integrity policies but also include investigations on how to these policies can be enforced.

Deliverable details

Deliverable version: *v1.0* | Classification: *public*
Date of delivery: *30.09.2011* | Due on: *30.09.2011*
Editors: *Daniel Hedin and Andrei Sabelfeld* | Total pages: *97*

List of Contributors:
Aslan Askarov, Nataliia Bielova, Arnar Birgisson, Bastian Braun, Dominique Devriese, Daniel Hedin, Peng Liu, Jonas Magazinius, Fabio Massacci, Frank Piessens, Joachim Posegga, Alejandro Russo, Andrei Sabelfeld

Project details

Start date: *October 01, 2010* | Duration: *36 months*
Project Coordinator: *Martin Johns*
Partners: SAP, Siemens, CHALMERS, KUL, UNI PASSAU

Executive Summary

The overall goal of the WebSand project is to empower web application developers, hosters, and users in designing, implementing, and using secure applications. As introduced in Deliverable 1.1, web applications share a number of properties that make them different from many other application domains. One example of such a property is that web applications are frequently built from a number of collaborating, but mutually distrusting components.

This deliverable reports on the results on information-flow security policies that have been accomplished by the project during the first year. To this end we investigate decentralized security policies for confidentiality and information release in the presence of mutual distrust. We present a framework that enables collaboration among different participants without compromising their respective security policies. A key feature is that information release is permitted only if the owners of the data agree on releasing it.

To ensure secure information release it is important that the decision when to release and what to release cannot be tampered with, i.e., cannot be influenced from the outside. This pertains to the notion of information integrity. Information integrity has important consequences for information security, but remains relatively uninvestigated. In this deliverable we explore different facets of information integrity and propose a unified framework for integrity policies.

Further, to lay the ground for Deliverable 3.3, we investigate a number of approaches to enforcement of the developed security policies including static type system based enforcement, enforcement via secure multi-execution, and dynamic enforcement via runtime monitors. These alternatives provide an excellent base for exploring whether the enforcement should be placed on client side, server side or as a collaboration between the two. On a related path we investigate the tradeoff between the expressiveness of security policies and enforceability. We present an information security policy — multi-run security — that can be seen as a middle ground between two common existing security policies.

Contents

List of Figures

1 Introduction

The overall goal of the WebSand project is to empower web application developers, hosters, and users in designing, implementing, and running secure applications. As introduced in Deliverable 1.1, web applications share a number of properties that make them different from many other application domains. One example of such a property is that web applications frequently are built from a number of collaborating, but mutually distrusting components. A typical example of this is the application paradigm of *web mashups* — a web service that integrates a number of independent services into a single web service. As an example of a common mashup consider a web application 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.

Mashup applications, by their nature, involve interaction between various page components of potentially different origin. Cross-origin interaction within the browser is currently regulated by the so-called Same-Origin Policy (SOP). The SOP classifies documents based on their origins. Documents from the same origin may freely access each other's content, while such access is disallowed for documents of different origins.

However, the SOP mechanism is insufficient to guarantee information security, since it allows only for binary "all-or-nothing" policies to be expressed, where different components are either prevented from accessing each others data or allowed full access. This leads to a conflict between the need to share information between the collaborating components and the need to protect the information. Since collaboration is impossible if the components are completely isolated from each other, the current solution is to allow full access at the expense of the confidentiality and integrity of the respective data.

Consider, for instance, a loan calculator provided by some financial institution, with the purpose that current and future customers can download the calculator, input their financial data and get details about the loans the financial institution can offer them. In order to enhance the loan calculator and future products the financial institution collects different statistics pertaining to the use of the application, e.g., the popularity of certain features of the loan calculator, and the relative popularity of the different loan setups.

This scenario contains a number of information-flow challenges. From the perspective of the user, it is important that the private financial information is not leaked to the statistics collected by the application.

Further, the loan calculator might be a part of a mashup providing a comparison of the offers of different financial institutions — c.f. best price services. From the perspective of the financial institution, it is important that the information provided by the user is not tampered with. Such tampering could for instance be used by external parties to skew the offers to favor a competitor.

A crucial challenge for building secure web applications is hitting the sweet spot between separation and integration, i.e., finding security policies that allow for controlling how information is shared between the mutually distrusting parts. It is this challenge that is tackled in the work package on information flow (work package 3) of the WebSand project.

This deliverable reports on the results on information-flow security policies that have been accomplished by the project during the first year. To this end we investigate decentralized security policies for confidentiality and information release in the presence of mutual distrust. We present a framework that enables combination of data owned by different participants 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.

This framework allows for securing the interaction in the scenario described above from the perspective of the user by making it possible to ensure that the financial information provided by the user is not leaked to the statistics without the consent of the user.

To ensure secure information release it is important that the decision when to release and what to release cannot be tampered with, i.e., cannot be influenced from the outside. This pertains to the notion of information integrity. Information integrity has important consequences for information security, but remains relatively uninvestigated. In this deliverable we explore different facets of information integrity and propose a unified framework for integrity policies.

This framework allows for securing the interaction in the scenario described above from the perspective of the financial institution by making it possible to ensure that the financial information provided to the loan calculator originates from the user, and has not been tampered with.

Further, to lay the ground for Deliverable 3.3, we investigate a number of approaches to enforcement of the developed security policies including static type system based enforcement, static enforcement via secure multi-execution, and dynamic enforcement via runtime monitors. These alternatives provide an excellent base for exploring whether the enforcement should be placed on client side, server side or as a collaboration between the two. On a related path we investigate the tradeoff between the expressiveness of security policies and enforceability. We present an information security policy — multi-run security — that can be seen as a middle ground between two common existing security policies.

Overview The rest of this section gives background on information-flow security and introduces the technical contributions of this deliverable. First, Section 2, based on [79], explores declassification in the setting of web applications that are built up by collaborating, mutually distrusting components. Thereafter, Section 3, based on [104], explores the tradeoff between expressiveness and enforceability for information-flow policies by investigating policies falling between two well-known extremes. This is followed by Section 4, based on [22], which explores practical enforcement of reactive noninterference by secure multi-execution.

Finally, Section 5, based on [24], explores a unifying model for integrity going beyond the standard interpretation of integrity as a dual to confidentiality.

1.1 Information flow

The research of *secure information flow* goes back to the early 70's [20, 48], primarily in the context of military systems. Secure information flow comprises of two related aspects: information *confidentiality* and information *integrity* — intuitively pertaining to the reading and writing of the information.

Confidentiality and integrity The prevailing basic notion of secure information flow for confidentiality is *noninterference* [58], demanding that the secret input of a program does not influence the public output. As the field has matured, numerous variations of noninterference [105], as well as other semantic characterizations have been explored [30, 28, 29, 10, 8].

Information integrity has received increasing attention [73, 74, 24, 7]. Information-flow integrity often means that trusted output is independent from untrusted input. This flavor of integrity is dual to the classical models of confidentiality, where public output is required to be independent from secret input.

Declassification and endorsement For many applications — e.g., applications built by collaborating components — such strong separation between secret and public information is too restrictive. Consider, for instance, a component that computes average salaries — even though each individual salary may be secret we might want to be able to share the average with other components.

Clearly, we need a way to *declassify* information, i.e., lowering the security classification of selected information. For integrity as the dual of confidentiality, the notion dual to declassification is *endorsement*, which allows for raising the integrity classification.

Aspects of information-flow policies There are two important aspects when designing security policies for secure information flow. First, different attack scenarios give rise to different attacker models which affects the demands placed on the security policies. Thus, the first aspect is capturing attacker models by adequate security policies. Second, for practical information-flow security, there is frequently a tradeoff between the expressiveness of the security policy and its enforceability. This means that policies must be rich enough to describe fine-grained security but at the same time it is desirable that these policies are amenable to automatic enforcement.

1.2 Confidentiality

1.2.1 From attacker models to confidentiality policies

Information flow security aims at protecting confidentiality and integrity of information. It is common to use a program-centric attacker model similar to the following: 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.

Deliverable 1.1 argues that for web applications a program-centric attacker model is not adequate. For collaborative distributed programs the classical Dolev-Yao attacker model [53] is frequently employed. In the Dolev-Yao model it is assumed that the attacker is able to overhear, intercept and modify messages on the network. In the web setting, however, a weaker notion of *web attacker* [17] 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. Hence, the web attacker is assumed to be unable to overhear, intercept or modify messages on the network, which implies that the web attacker is unable to mount man-in-the-middle attacks.

1.2.2 Decentralized delimited release

Information flow security for programs built from collaborating, mutually distrusting parts requires decentralized security policies. Decentralization is a major challenge for secure computing. The key challenge is to provide possibilities for expressing *and* enforcing expressive decentralized policies, that allows principals to trust or distrust other principals.

Further, collaboration requires not only protection of information, but also the ability to share information. Section 2 focuses on decentralized *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.

1.2.3 Multi-run security

For practical information flow security, however, the choice of security policy can not be decided based on the capabilities of the attacker alone. In addition, there is a tradeoff between the expressiveness and precision of the security policy, and the enforceability. The state of the art in the area of policies for information flow security consists of two extremes described in Deliverable 1.1.

On one side are batch-job program models corresponding to termination insensitive noninterference, where programs are run only once and where the initial memory is the only input and the final memory is the only output. While securing batch-job programs without being over-restrictive is feasible, the assumption that programs are run only once is often too strong.

On the other side are fully interactive programs with channels for input/output communication corresponding to progress sensitive noninterference. While this model is more powerful, securing interactive programs is notoriously hard: intermediate observations can be exploited to leak information [6].

For this reason it is interesting to explore the territory for intermediate models. Section 3 studies information-flow control for batch-job programs that are allowed to be re-run with new input provided by the attacker. We argue that directly adapting two major security definitions for batch-job programs, termination-sensitive and termination-insensitive noninterference, to multi-run execution would result in extremes. While the former readily scales up to multiple runs, its enforcement is typically over-restrictive. The latter suffers from insecurity: secrets can be leaked in their entirety by multiple runs of programs that are secure according to batch-job termination-insensitive noninterference. Seeking to avoid the extremes, we present a framework for specifying and enforcing multi-run security in an imperative language. The policy framework is based on tracking the attacker's knowledge about secrets obtained by multiple program runs. Inspired by previous work on robustness, the key ingredient of our type-based enforcement for multi-run security is preventing the dangerous combination of attacker-controlled data and secret data from affecting program termination.

1.3 Integrity

Where declassification 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.

1.3.1 From attacker models to integrity policies

The most prevalent facet of integrity is when integrity is taken to mean that information can be trusted if it cannot be influenced by the attacker. Phrased in terms of information flow, trusted output should be independent from untrusted input. In this interpretation standard models used for confidentiality suffice.

Integrity in the area of access control [111] 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 [111, 98].

Sabelfeld and Myers [105] observe that integrity has an important difference from confidentiality: a computing system can damage integrity without any external interaction, simply by computing data incorrectly. (This can happen as a consequence of either programming errors or system faults.) Thus, strong enforcement of integrity requires proving program correctness. Seeking to clarify the area of integrity policies, Li et al. [73] suggest a classification for data integrity policies into information-flow, data invariant and program correctness policies. In a similar spirit, Guttman [60] identifies causality and invariance policies as two major types of data integrity policies. Furthermore, we argue that integrity via invariance is itself multi-faceted. For example, the literature (cf. [73]) features formalizations of invariance as predicate preservation (predicate invariance), which is not directly compatible with invariance of memory values (value invariance).

1.3.2 Unifying facets of information integrity

Section 5 offers a unified framework for integrity policies that include all of the facets above. Despite the different nature of these facets, we show that a straightforward enforcement mechanism adapted from the literature is readily available for enforcing all of the integrity facets at once.

1.4 Enforcement

Deliverable 1.1 presents two fundamentally different approaches to enforcing secure information flow: *static*, i.e., before running the program, and *dynamic*, i.e., during the running of the program. The choice frequently amounts to a tradeoff between efficiency and precision, but certain programming language features may pose more fundamental obstacles for either static or dynamic approaches. Regardless, *hybrid* analyses, i.e., a combination of both static and dynamic enforcement may be used to increase precision. For web application the additional aspect of where enforcement takes place — on the server, on the client, or in collaboration between the two – is important.

With respect to the policies presented above, we explore both static and dynamic enforcement. The policies of Section 3 are enforced by a static type system, which prevents looping on expressions that both depend on secrets and attacker input. At the other end, the policies of Section 2 and Section 5 are both enforced dynamically via runtime monitors. Further, we develop a hybrid version of the mechanism from Section 2, where the dynamic monitor is inlined into the code of a given program using static program analysis and transformation.

Section 4 explores an enforcement technique for reactive non-interference based on secure multi-execution [50]. It is shown that Featherweight Firefox [25] (in fact any reactive system in the sense of Bohannon et al. [26]) is reactive non-interferent when executed under this secure multi-execution regime, and that, for inputs for which Featherweight Firefox is "well-behaved" with respect to the policy, execution under the secure multi-execution regime will not result in changes in observable behavior for an observer at any security level.

In addition, the value for web browsers of the technique is shown, by implementing it for Featherweight Firefox. To the best of our knowledge, this proposal is the first to enforce a general non-interference policy for the browser as a whole.

Finally, this implementation of a secure and precise enforcement mechanism for a model browser allows us to experiment with suitable policies. We investigate three classes of policies, ranging from simple policies that can provide high-assurance confidentiality guarantees for user-private data to more complicated policies that support common web-patterns like third-party library script inclusion while maintaining strong restrictions on the flow of information between different origins.

2 Decentralized delimited release

2.1 Introduction

Decentralization is a major challenge in particular 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 [105]. 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, in the sense that the flow of information is tracked through the entire system from information sources to sinks. For example, a common security goal is *noninterference* [44, 58, 122, 105] 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 [105], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [92, 114, 118].

A particularly important problem in the context of information-flow security is *declassification* [110] 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 2.5), 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 [46] 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 [78].

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)* [88]. This model underlies the security labels tracked by the Java-based information-flow tracker Jif [92]. 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* [106], this work 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.

2.2 Decentralized delimited release

Principals and security levels Our model is built upon a notion of *principals* which we denote via p, q. We assume that principals are mutually distrusting and that there are no “actsfor” or “speaks-for” relations [89, 70] 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 2.2.2.

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 [2, 108]. The PER representation allows for fine granularity in individual policies. We believe that intentional models of security such as DLM [89] or tag-based models [54, 69, 126, 35] can be easily interpreted using PERs. Section 2.2.2 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* $Id \triangleq \{(m, m) \mid m \in M\}$ *and* $All \triangleq \{(m, m') \mid m, m' \in M\}$

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* $\mathsf{Ind}(\mathcal{E}) \triangleq \{(m, m') \mid \forall e \in E \,.\, m(e) = m'(e)\}$.

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 useful in the following examples and in the translation in Section 2.2.1.

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 $I(L) = \mathit{All}, I(M) = \mathsf{S}(x)$, and $I(H) = \mathit{Id}$

According to this policy, an attacker at level L can observe no part of the state, which is specified by $I(L) = \mathit{All}$. 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 $I(\top) = \mathit{Id}$, 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, $I(H) = \mathit{Id}$ and $I(L) = \mathit{All} \supseteq I(M) = \mathsf{S}(x) = \mathsf{Ind}(y) \supseteq I(H) = \mathit{Id}$. 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 work assumes that all policies are well-formed.

2.2.1 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 $p_1 \dots p_n$ 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* [106] 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 $\mathcal{E}_\ell = \{e \mid (e, \ell') \in E \wedge \ell' \sqsubseteq \ell\}$ is the selection of escape hatches from $\mathcal{E}$ that are observable at 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	$\mathsf{S}(x) \cap \mathsf{Ind}(x)$	$\mathsf{Ind}(x+y)$

Figure 1: Declassification and composite policies

Example: Consider I_p as in Section 2.2.1 and escape hatch (y, L). Let us assume $I' = D(I_p, \{(y, L)\})$. We have $I'(L) = \mathsf{Ind}(x)$, $I'(M) = \mathit{Id}$, and $I'(H) = \mathit{Id}$.

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 $\mathsf{Ind}(y) \cup \mathit{All} = \mathit{All}$. That is, principals agree on no information being observable to an adversary at the level L.

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 2.3 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 [106], and generalizes it to the case of several principles. In fact, in case of a single principal this definition matches the original definition in [106].

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) [88] as our model of expressing security policies sans $\mathsf{actsfor}$ relation, that we dub DLM^0 . We nevertheless, retain top and bottom principals $\perp$ 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 $o \rightarrow r_1, \ldots, r_n$, 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 $\mathit{Alice} \rightarrow \mathit{Bob}, \mathit{Carol}$ Alice constraints the set of readers to only Bob, Carol, and herself (the owner is implicitly a reader). Similarly, a policy $\mathit{Carol} \rightarrow \mathit{Carol}$ 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,

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	$\mathsf{S}(x, y)$	$\mathsf{S}(x, y)$	$\mathsf{S}(x, y) \cap \mathsf{Ind}(x + y)$	$\mathsf{S}(x, y) \cap \mathsf{Ind}(x) \cap \mathsf{Ind}(y)$

Figure 2: Policies for example laundering attack

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" [90, 38] 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 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.

2.2.2 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 $\Gamma = \emptyset$ 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*
$$I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup S(x) & \text{if } \{q \to \vec{r}\} \not\sqsubseteq \ell'' \\ I_p(\ell'') & \text{otherwise} \end{cases}$$
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* $I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup S(x) & \textit{if } \{q \to \vec{r}\} \not\sqsubseteq \ell'' \wedge \ell' \not\sqsubseteq \ell'' \\ I_p(\ell'') & \textit{otherwise} \end{cases}$ *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* $\widetilde{\mathsf{T}}_p(\emptyset) = \widetilde{Id}$*, when* $\Gamma = \emptyset$*, and otherwise* $\widetilde{\mathsf{T}}_p(x \mapsto \ell; \Gamma') = T_p(\widetilde{T}_p(\Gamma'), \ell, x)$*. 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 $\mathcal{E}_p = E_q = \{(x + y, \{p \to p, q\} \sqcup \{q \to p, q\})\}$. Taking escape hatches into account the policies obtained from declassification operator are illustrated in fourth and fifth columns of the table above.

2.3 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 non-standard primitive in the language is a declassification primitive `declassify`(p, e, ℓ) that declares an escape hatch (e, ℓ) of principal p.

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

Figure 3: Syntax

$$\langle \texttt{declassify}(p, e, \ell), m\rangle \xrightarrow{d(p,e,\ell)} \langle stop, m\rangle$$

$$\frac{m(e) = v}{\langle x := e, m\rangle \xrightarrow{a(x,e,m)} \langle stop, m[x \mapsto v]\rangle}$$

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

$$\langle end, m\rangle \xrightarrow{f} \langle stop, m\rangle$$

Figure 4: Monitored semantics: selected rules

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

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

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

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

Figure 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., [107, 10]). 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 [78], 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 2.5 discuses these and other dimensions of declassification [110] 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 [12].

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 $\bot \rightarrow \bot$.

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* [49] 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 $\mathsf{substEH}$ 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 $\mathsf{substEH}$ 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 `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 2.4) 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 $\bigcup_p \widetilde{\mathsf{T}}_p(\Gamma'_1)(\ell) = \bigcup_p \widetilde{T}_p(\Gamma'_2)(\ell)$ 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 [80].

Example: We revisit the example with aggregate computation from Section 2.2.1. Consider variable environment consisting of three variables x, y and z. Assume two principals p and q s.t. $\Gamma(x) = \{p \rightarrow p\}, \Gamma(y) = \{q \rightarrow q\}$, and $\Gamma(z) = \{p \rightarrow p, q\} \sqcup \{q \rightarrow p, q\}$. and escape hatch sets for every principal s.t. $\mathcal{E}_p = E_q = \{(x + y, \{p \rightarrow p, q\} \sqcup \{q \rightarrow p, q\}, \bot \rightarrow \bot)\}$. Then basic declassification of the form $z := x + y$ is accepted, while laundering as in the program $x := y; z := x + y$ is rejected.

2.4 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 2.3 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 [5] 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 [95] 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 [82] 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., [81]. 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 2.3. 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 2.3, 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 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.

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

Listing 1: Naming convention

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 piggy-back 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.

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 2: Assignment rule

Insecure upgrade refers to assignment of a lower level variable in a higher level context, implying an information leak [101]. Insecure upgrade is prevented by checking that the `_pc` is less than or equal to the level of the variable [12]. 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 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).

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

```
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 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 4: Scenario 2 transformed

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 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 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.

2.5 Related work

There is a large body of work on declassification, much of which is discussed in Sabelfeld and Sands' recent overview [110]. 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 [83, 9, 14, 18, 10].

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) [88]. DLM allows principals expressing ownership information as well as explicit read/write access lists in security labels. Chen and Chong [33] generalizes DLM to describe a range of owned policies from information flow and access control to software licensing.

Work on *robustness* [91, 7], addressed the *who* dimension by preventing attacker-controlled data from affecting *what* is released. Lux and Mantel [76] 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* [78] 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 work removes this limitation and generalizes the principal model to handle DLM. The experimental part is another added value with respect to the previous work [78].

Broberg and Sands [30] 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 [78], where we present considerations for practical enforcement of composite release. The formalization of the enforcement fits well into the modular framework [107, 10] 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 [10], by ensuring that the values of escape-hatch expressions have not been modified. This work extends the formalization of the enforcement with the *who* part.

Recent efforts approach inlining for information flow. Chudnov and Naumann [40] inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [101]. The monitor does not offer support for declassification. As in this work, Magazinius et al. [82] 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 [82] offer no support for declassification, it is still a useful starting point for our experiments in Section 2.4.

In the web setting, work on language-based sandboxing such as object capabilities (e.g., [87, 77]) is less related because separation does not allow information flow and intended release. The most closely related project is the Mozilla project FlowSafe [55] that aims at extending Firefox with runtime information-flow tracking, where dynamic information-flow monitoring [12, 13] lies at its core.

2.6 Conclusion

We have presented a framework for specifying 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 [10], timeout events [102], and interaction with the DOM tree [103].

Another intriguing avenue for integration is with Chong's *required release* [36] 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.

3 Multi-run security

3.1 Introduction

Imagine a scenario of a web service with a medical database at the back-end. Analysts are allowed to access the database through a web interface. The goal is to allow deriving interesting statistics (say, by age groups or by larger residential areas) but disallow leaking sensitive information about individuals. In this scenario, the server-side program that accommodates queries has two inputs: one is the database itself, which contains sensitive data and which is not controlled by the attacker, and the other one is a public query that originates from a possibly malicious analyst. For the program to function, it must have access to the entire database. At the same time, it must not reveal sensitive data about individual entries in the database. Hence, we are interested in securing *information flow* from secret inputs to public outputs. This problem arises both when the code is written by non-malicious developers, in which case we want prevent accidental leaks, and when the code is supplied by untrusted third parties, when we want to prevent malicious leaks. Settling for the worst case, we do not appeal to trust assumptions.

Language-based information-flow security [105] is focused on providing strong security guarantees for underlying programs. In the context of confidentiality, it is intended to prevent information flow from secret inputs to public outputs. The dominating baseline security policy is *noninterference* [44, 58] that requires that a variation of secret input does not result in a variation of public outputs.

However, the state of the art in the area consists of two extremes. One extreme is batch-job program models, where programs are run only once and where the initial memory is the only input and the final memory is the only output. A large body of research on language-based information-flow security is limited to batch-job models. In a language-based setting, noninterference has been largely considered for batch-job models [122, 105]. Major efforts on information flow in functional [99], object-oriented [127, 15, 62], concurrent [117, 125, 101], and other languages [105] assume a batch-job model.

While securing batch-job programs without being over-restrictive is feasible, the assumption that programs are run only once is often too strong. The other extreme is fully interactive programs with channels for input/output communication. While this model is more powerful, securing interactive programs is notoriously hard: intermediate observations can be exploited to leak information [6].

This section explores middle ground between the extremes: batch-job programs that are allowed to be re-run with new input provided by the attacker. We believe this model captures many practical scenarios such as the medical database above. Our attacker model allows issuing queries to the database as described by a batch-job program whose secret input is the database and public input is the attacker-controlled part of the query. The goal is to prevent the attacker from learning sensitive information by re-running the program with modified public parameters and observing the public outcome.

Leaks via termination behavior of programs turn out to be the bottleneck for generalizing batch-job style security to multiple runs. We argue that directly adapting two major security definitions for batch-job programs, termination-sensitive and termination-insensitive noninterference, to multi-run execution would result in further extremes. The former, *termination-sensitive noninterference* [121, 105], readily scales up to multiple runs. This definition demands that the public outcome and termination behavior of the underlying program do not depend on secret data. No run leaks any information about secrets, and so we can safely re-run programs. Thus, batch-job termination-sensitive security implies multiple-run security. However, enforcing termination-sensitive noninterference without being overly restrictive is far from trivial because it requires tracking abnormal termination and divergence. Typically, enforcement mechanisms (e.g., [121]) place Draconian restrictions whenever abnormal termination is possible in sensitive context. For example, no sensitive data is allowed in loop guards.

The latter, *termination-insensitive noninterference* [122, 105], where secrets are allowed to affect termination behavior, suffers from insecurities in the multi-run case. Let us illustrate the problem with examples. The program

$$\texttt{while } h \texttt{ do skip} \tag{1}$$

where h contains a secret, is deemed secure. Termination-insensitive noninterference quantifies over all possible input memories that agree on the public part and makes sure that terminating runs agree on the public part of the final memories. The termination behavior is not considered to have a significant effect, even though the termination depends on secret data. Although the condition quantifies over possible runs, its guarantees are only about differences between two single runs. The implicit assumption is that the program is run only once. A common argument is that if a batch-job program that satisfies termination-insensitive noninterference is run only once, then it leaks at most one bit [6].

With the same rationale, flavors of this program are also accepted by mainstream information-flow security tools Jif [92], FlowCaml [114], and the SPARK Examiner [16, 32] for Java, Caml, and Ada, respectively.

Similarly, the program

$$\texttt{while } h = l \texttt{ do skip} \tag{2}$$

where h contains a secret and l is an attacker-controlled public variable, is also considered secure.

However, the single-run assumption is in many cases inadequate. As in the database scenario above, attackers are often capable of re-running the program. Further, in a smartcard setting, the attacker may try to leak the secret key by multiple attempts of feeding public inputs and observe the properties of public outputs. A web attacker can initiate multiple runs of a server-side computation that involves secrets by providing a request with public input. Similarly, the attacker can initiate multi-run computation on the client side of an honest user by providing scripts that keep re-running after recovering from divergence (rather straightforward to accomplish with the modern browsers' interpretation of JavaScript). A recent exploit of ASP.NET analyzes the difference between error messages of multiple requests to collect information for a padding oracle attack [100].

This ability does not make a difference for program 1 (given the value of the secret is unchanged between the runs), but it is fatal for program 2: the attacker can learn the entire secret by brute-force guessing the value of h with different choices for l. Multi-run leaks are particularly devastating for single-run secure programs like:

$$\texttt{while}\ h\&\&l\ \texttt{do skip} \tag{3}$$

where $\&\&$ is the bitwise "and" operation. By walking through the bits of h in subsequent runs, the attacker can learn the entire value in linear time (of the bit-size of the secret) bit-by-bit. Thus, secrets can be leaked in their entirety by multiple runs of programs that are single-run secure. A quick experiment with a Jif-certified program that contains a termination leak of this kind shows that it is straightforward to leak one secret bit per second even on a modest modern desktop machine (tested with Jif 3.0). This implies that a credit card number can be leaked within a minute. The Jif program and a simple Python script that exploits its termination leak are shown in Appendix B of [104].

Seeking to avoid the extremes, we present a framework for specifying and enforcing multi-run security. For specification, we are inspired on knowledge-based attacker models [51, 8]. The policy framework is based on tracking the attacker's knowledge about secrets obtained by multiple program runs. The multi-run setting for such a framework is novel. The framework supports possibilities for intended information release (illustrated by examples below). Further, it connects to quantitative security, where we reason about how many bits of information can be leaked by multiple program runs.

For enforcement, we are inspired by previous work on robustness [124, 91, 7]. The key ingredient of our type-based enforcement for multi-run security is preventing the dangerous combination of attacker-controlled data and secret data from affecting program termination. It is particularly gratifying that we can draw on the type system for robustness for enforcing a policy that it has not been designed for. This connection leads us to clean enforcement, providing a simple solution to a nontrivial problem of multi-run security.

For information-flow tracking, we deploy data labels that combine confidentiality and integrity information. Confidentiality distinguishes secret information from public by *high* and *low* confidentiality labels. Integrity distinguishes untrusted information from trusted by *low* and *high* integrity labels. For confidentiality the use of high information is more restrictive: secrets may not leak to public; and dually for integrity use of low is restricted: untrusted data may not affect trusted. Typically, lattices [48] are used to reason about more complex structures than low/high for confidentiality and integrity. Of particular interest to us are product lattices that combine confidentiality and integrity labels. In the example of a product lattice that combines two low/high lattices, the top element is high confidentiality and low integrity. Data at this level is most restrictive to use. The bottom element is low confidentiality and high integrity, which may arbitrarily affect data at other levels. Integrity plays a key role for the enforcement: the enforcement ensures that combinations of high-confidentiality (secret) and low-integrity (attacked-controlled) data do not affect the termination behavior.

As foreshadowed above, we extend our approach to specify and track intentional information release (or *declassification*). The extended enforcement guarantees that the program does not release more information than described by *escape-hatch* [106] expressions. The purpose of escape hatches is to describe what is allowed to be released. The job of the underlying security condition is to ensure than *nothing else* about secret data may be learned by the attacker. For example, program

$$l := h\%4 \tag{4}$$

releases two least-significant bits about the secret variable h. When this is desired, it is expressed in our framework by the escape-hatch expression $h\%4$. The type system accommodates intentional release by labeling escape-hatch expressions as low confidentiality and high integrity. Hence, the program above is accepted while, for example, program

$$l := h\%6 \tag{5}$$

is rejected because the type system detects a mismatch with the escape hatch $h\%4$.

Next, assuming the same escape-hatch policy $h\%4$, consider the following program:

$$\texttt{while } h\%4 + l \texttt{ do skip} \tag{6}$$

This program may also release two least-significant bits about h. Indeed, the attacker may experiment by supplying inputs -2, -1, and 0 for l and observing whether the program diverges. Our type system rightfully accepts this program because the loop guard $h\%4{+}l$ has low integrity and low confidentiality, inheriting its restrictions from variable l, which has low integrity, and $h\%4$, which has low confidentiality.

A final example illustrates how intended declassification is distinguished from unintended. Assuming the escape-hatch policy h, consider the program

$$h := h'; l := h \tag{7}$$

that attempts to leak the initial value of h' by laundering its value through the declassified (syntactic) variable h. This program is rejected because the enforcement mechanism detects that a variable involved in declassification has been modified.

3.2 Security condition

This section presents some key definitions, in particular the definition of when we consider programs multi-run secure. Command c represents a deterministic program in the rest of this section. As before, h and l represent secret (*high*) and public (*low*) variables. Without loss of generality, we treat a program as a function of two inputs (secret and public) coming from some finite domain D, to the set $D \cup \{\perp\}$. The result $c(h,l)$ expresses the observed low output of the program, with the special value $\perp$ representing nontermination.

Definition 1 (Single-run knowledge) *Let c be a program taking two inputs, a fixed secret one v_h and a (non-fixed) public one v_l each from some domain D, and yielding a public output $c(v_h, v_l) \in D \cup \{\perp\}$.*

An attacker (with full knowledge of c itself) is allowed to execute c, providing the public input v_l and observing only the public output $c(v_h, v_l)$. The attacker's knowledge of the (fixed) secret input is then represented by the set of values that would lead to the observed outcome. This set is written as:

$$k_{v_h}(c, v_l) = \{\, x \in D \mid c(x, v_l) = c(v_h, v_l) \,\}$$

Programs with more than two inputs are modeled by collecting all secret and public inputs into two separate tuples, and similar for programs which have more than one output.

Note that by allowing $c(v_h, v_l)$ to take the special value $\perp$ (meaning that c has an infinite derivation for those inputs), we make nontermination observable. This is important because in reality, nontermination can for example be (approximately) inferred from programs that time out.

Assume an attacker has some previous knowledge of h, represented by the set $K_0 \subseteq D$. Then the attacker is potentially able to increase that knowledge (which corresponds to shrinking the set of possibilities) by running the program. The attacker's new knowledge will be the single-run knowledge intersected with the previous knowledge. Repeating this process (possibly with different low inputs) results in a sequence of increasing knowledge (decreasing sets of possibilities). For an attacker with no initial knowledge we can simply start with $K_0 = D$. Obviously, a program may potentially leak more if the attacker gets the chance to invoke it multiple times and has control over some of the input.

The maximum knowledge attainable by the attacker is the result of the above process repeated for every possible low input. Note that this models a powerful attacker, as the number of possibilities is exponential in the bit-size of the input. This maximum knowledge, or multi-run knowledge is now defined as follows.

Definition 2 (Multi-run knowledge) *Let c, v_h, v_l, and D be as in Definition 1. The attacker's knowledge about v_h produced by multiple runs of program c is defined as:*

$$K_{v_h}(c) = \bigcap_{v_l \in D} k_{v_h}(c, v_l)$$

To highlight the contrast between multi-run knowledge and single-run knowledge captured by the definitions, we come back to the examples from Section 3.1. Assume $D = \{0, \ldots, 255\}$. Recall program 1:

$$\texttt{while } h \texttt{ do skip}$$

The single-run knowledge $k_{v_h}(c, v_l)$ for this program is $\{0\}$, when $c(v_h, v_l) = v_l$, and $\{1, \ldots, 255\}$, when $c(v_h, v_l) = \perp$. The multi-run knowledge $K_{v_h}(c)$ is $\{0\}$, when $v_h = 0$, and $\{1, \ldots, 255\}$, when $v_h \neq 0$, which directly corresponds to the two cases for the single-run knowledge. Recall now program 2:

$$\texttt{while } h = l \texttt{ do skip}$$

The single-run knowledge $k_{v_h}(c, v_l)$ for this program is $\{0, \ldots, v_l - 1, v_l + 1, \ldots, 255\}$, when $c(v_h, v_l) = v_l$, and $\{v_l\}$, when $c(v_h, v_l) = \perp$. However, the multi-run knowledge $K_{v_h}(c)$ is simply $\{v_h\}$, which corresponds to leaking all of v_h into variable l. The intersection in the definition of multi-run knowledge corresponds to traversing all possible low inputs in the attempt to match them to v_h, which is a worst-case model for multi-run attackers.

Now that we have definitions of attacker knowledge obtained after running the program, we wish to express a policy which sets limits on this knowledge. A knowledge policy states a lower bound on the attacker's uncertainty by partitioning the input domain into classes. Each class lists values that must remain indistinguishable to the attacker. In other words, the attacker may identify from which class the secret input comes, but any more precision is disallowed. This view corresponds to *partial release* [44, 109] of information. This leads to the following definition.

Definition 3 (Knowledge-policy) *A knowledge policy P for an input with domain D is a partition of D into classes P_i:*

$$P = \{P_1, \ldots, P_n\} \qquad P_i \subseteq D \qquad i \neq j \implies P_i \cap P_j = \emptyset \qquad P_1 \cup \ldots \cup P_n = D$$

For a value $v \in D$ we write $[\![v]\!]_P$ to represent the class of P to which v belongs.

Note that as a partition of D, a policy represents an equivalence relation on values. Two values are equivalent if they come from the same class. This equivalence relation is often referred to as an indistinguishability relation [44, 109].

We illustrate the definition with simple examples. A policy that allows the attacker no knowledge is simply $P = \{D\}$. A policy that allows full knowledge is $\{\{x\} \mid x \in D\}$. If D is the set of unsigned 8-bit integers, then a policy that allows the attacker to know the parity of the secret is:

$$\{\{0, 2, \ldots, 254\}, \{1, 3, \ldots, 255\}\}.$$

We are now ready to state the formal definition of multi-run secure programs.

Definition 4 (Multi-run security) *Let c be a program that takes a secret input v_h and an arbitrary public, attacker-controlled input. c is multi-run secure (or simply secure) with respect to a knowledge policy P if and only if $[\![v_h]\!]_P \subseteq K_{v_h}(c)$.*

Observe that multi-run security with policy $\{D\}$ corresponds to *termination-sensitive noninterference* [121, 105] for single runs, which prevents the termination behavior of the program (as well as its public output) from being affected by secrets.

Recall programs 4 and 5 from Section 3.1. Program 4 ($l := h\%4$) is secure for all high input with respect to the policy $P = \{\{0, 4, \ldots\}, \{1, 5, \ldots\}, \{2, 6, \ldots\}, \{3, 7, \ldots\}\}$. Indeed, the multi-run knowledge from running the program has to be one of the four sets in the policy because the attacker only learns the two least-significant bits.

On the other hand, program 5 ($l := h\%6$) is insecure for all high input according to the policy P. To illustrate this, take $v_h = 0$. The multi-run knowledge $K_0(c)$ is $\{0, 6, \ldots\}$, while $[\![0]\!]_P = \{0, 4, \ldots\}$ which is clearly not contained in the knowledge.

As we are interested in an enforcement mechanism that allows limited leaks through the termination channel, Definition 4 will serve as the basis for a relaxed definition which we apply to the enforcement mechanism presented in Section 3.3. This relaxation draws on ideas from quantitative security. Smith [116] defines the notion of *vulnerability* $V(X)$, which is the worst-case probability of guessing the value of secret X by an adversary in one try. The measure of information quantity is then defined as $-\log V(X)$. Based on the intuition "information leaked = initial uncertainty - remaining uncertainty", Smith defines *information leakage* and shows that for deterministic programs and uniformly distributed secrets it amounts to $\log |S|$, where $|S|$ is the size of the set of possible public outputs given the public input is fixed. The intuition is that the more different observations the attacker can observe, the more secret information about might leaked through them. In the multi-run case, the size of the set of possible outputs translates to the number of indistinguishability classes for the high input, which, in effect, is the number of different values $K_{v_h}(c)$ can take when v_h varies. This is in line with Lowe [75], who measures the number of secret behaviors distinguished by an attacker in a nondeterministic setting. This motivation brings us to the following definition of security of programs that operate on uniformly distributed secrets:

Definition 5 (k-bit security) *Let c be a program that takes a uniformly distributed secret input v_h and an arbitrary public, attacker-controlled input v_l. c is $\underline{k\text{-bit secure}}$ if $k = \log n$ and $K_{v_h}(c)$ takes at most n distinct values as v_h varies.*

For example, program 1 is 1-bit secure because there are only two possibilities for $K_{v_h}(c)$ as v_h varies. On the other hand, program 2 is k-secure, where k is the bit size of h because $K_{v_h}(c)$ ranges over all possible singleton sets as v_h varies.

1-bit security is a particularly interesting case. Intuitively it means that an attacker can at most infer that v_h is in some set A or that it is in A's complement. In an extreme case, either set might contain only one element, meaning the attacker would know the exact value $\underline{\text{of that particular } v_h}$, but since there are only two possible "knowledges" this is equivalent to the attacker being allowed only one boolean test on the secret.

Ultimately we will prove that our enforcement mechanism is multi-run secure with respect to a policy, with the relaxation that 1-bit leaks are allowed. For simplicity we combine Definition 4 and the 1-bit version of Definition 5 as follows.

Definition 6 (1-bit security w.r.t. a policy) *Assume c, v_h, v_l are as in Definition 5, and P is a knowledge policy. We say that c is $\underline{\text{1-bit secure with respect to } P}$ if and only if for each class $P_i \in P$, $K_{v_h}(c)$ takes at most two distinct values (knowledges) K_1, K_2 as v_h varies within P_i, and furthermore $P_i \subseteq K_1 \cup K_2$.*

In other words, $K_{v_h}(c)$ can vary arbitrarily for v_h from different indistinguishability classes, but within each class we only allow for revealing one additional bit of

$$\begin{aligned}
&n \in D, \quad x \in \mathit{Vars}, \quad \mathit{op} \in \{+, -, \dots\} \\
&e ::= n \mid x \mid e\,\mathit{op}\,e \\
&c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c
\end{aligned}$$

Figure 6: Syntax

information. The last part ensures that an attacker cannot otherwise exclude any values from the policy class of the secret, any value considered impossible in one knowledge must be considered possible according to the other knowledge.

3.3 Enforcement

We illustrate our approach to enforcement for an imperative language. To keep the exposition clear, we have deliberately chosen a simple language, but the ideas here scale to more complex languages. Figure 6 shows the syntax of the language. Expressions take literals from a finite domain D (e.g., 32-bit integers) and variables from a set Vars. We present a type system for this language such that typable programs are robust against multi-run attacks that try to magnify single-run termination leaks into leaking more than one bit. The type system represents a static analysis, conveniently referring to security labels for variables and expressions as *security types*.

We will continue to treat programs as functions $D \times D \to D \cup \{\perp\}$, and in concrete examples the inputs will be represented by the variables h and l. The final value of l will be the output for terminating programs.

The important feature of this type system is that it does not allow looping on expressions that both depend on secrets and attacker input. Thus we need to consider both the confidentiality and integrity levels of expressions at the same time. To achieve this we label variables with labels from the following product lattice $\mathcal{L}$ that combines confidentiality and integrity.

$$\begin{tikzcd}
 & HL \arrow[dl, dash] \arrow[dr, dash] & \\
HH \arrow[dr, dash] & & LL \arrow[dl, dash] \\
 & LH &
\end{tikzcd}$$

Here a label lists first the confidentiality level and then the integrity level. For example, the attacker provided input l has level LL (low confidentiality, low integrity) since it is both known and controlled by the attacker, and the secret input h has level HH since it is neither. An expression combining a secret with untrusted input will be assigned level HL (high confidentiality, low integrity). We use the standard symbols $\sqsubseteq, \sqcup$, etc. for lattice operators. This lattice has been used for enforcing *robust declassification* [124, 91, 7], which demands that the attacker may not affect what is released by programs by ensur-

$$\text{SKIP}\frac{}{pc \vdash \texttt{skip}} \qquad \text{ASSIGN}\frac{lev(e) \sqcup pc \sqsubseteq lev(x)}{pc \vdash x := e}$$

$$\text{SEQ}\frac{pc \vdash c_1 \qquad pc \vdash c_2}{pc \vdash c_1; c_2} \qquad \text{IF}\frac{pc \sqcup lev(e) \vdash c_1 \qquad pc \sqcup lev(e) \vdash c_2}{pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}$$

$$\text{WHILE}\frac{pc \sqcup lev(e) \vdash c \qquad pc \sqcup lev(e) \neq HL}{pc \vdash \texttt{while}\ e\ \texttt{do}\ c}$$

Figure 7: Typing rules

ing that only high-integrity data can be declassified, and only in a high-integrity context. The work on robust declassification is a direct inspiration for our treatment of the termination channel in multi-run security. However, as we explain in Section 3.6, the policy that robust declassification enforces is rather different from our security model. Our observation that connects robust declassification with multi-run security enables us to cleanly reuse the enforcement technique, but still requires us to show soundness with respect to our security goals.

3.4 Enforcing 1-bit security

We start by showing that with a simple type system, we can make sure that typable programs cannot be used to magnify termination leaks beyond the traditional one-bit limit. The core idea is that the type system prevents information that is a mix of secrets and untrusted inputs from affecting termination behavior, by disallowing it in loop guards.

We equip the set of variables with a function giving the label of each variable, $label : Vars \to \mathcal{L}$. For expressions in general we define the function lev, assigning each expression with its security level. Function lev is defined as follows, pattern matching on the form of expression:

$$lev(n) = LH \qquad lev(x) = label(x) \qquad lev(e_1\ op\ e_2) = lev(e_1) \sqcup lev(e_2)$$

While variables have their corresponding label as a level, literals are always low confidentiality and high integrity, as we assume the program source to be public but trusted. Other expressions take the least upper bound of their component levels.

Figure 7 gives the typing relation. The typing context consists only of the level of the program counter, pc. This level represents expressions on which the control flow context depends, namely `if` and `while` guards. If a command c

is typable under context pc, written $pc \vdash c$, the intention is that c does not leak when executed, even if the execution itself is conditioned on data of level pc or higher. Branches of an if-command must be typable under the outer pc level joined with the level of the guard expression (rule IF). The rule ASSIGN uses this to prevent implicit flows: assignments to a variable are only allowed when both the expression and the program counter are below or at the same level as the level of the assigned variable.

The rule WHILE propagates the level of the guard in the same way as IF, but in addition requires that the guard expression joined with the context pc is strictly below HL. The intention here is to prevent the attacker from selectively inducing nontermination that depends on the secret.

For example, program 1

$$\texttt{while}\ h\ \texttt{do skip}$$

is typable, because the level of the guard is HH. As we show below, this implies that it only leaks one bit and the attacker is not able to change termination behavior by varying the public input. Same goes for program

$$\texttt{while}\ l\ \texttt{do skip}$$

since although the attacker can control termination, it does not reveal anything about the secret. The level of the guard here is LL. However program 2

$$\texttt{while}\ h = l\ \texttt{do skip}$$

is not typable, as the level of the guard is HL. Indeed, recall that the attacker is able to try different inputs until one is found that corresponds to the secret, in which case the whole of h is revealed.

Our goal is to prove that the type system enforces that programs leak at most one bit (via a termination leak) even in the multi-run setting. To prove that typable programs leak at most one bit, we will start by excluding leaks other than termination leaks. This means that terminating programs satisfy noninterference, i.e., the observable output is independent of the secret input. First, we show that programs typable with a high-confidentiality pc cannot modify the low output.

Lemma 1 *Let c be a program. If $HL \vdash c$ or $HH \vdash c$, then for any choice of $v_h, v_l \in D$, if $c(v_h, v_l) \neq \perp$ then $c(v_h, v_l) = v_l$.*

The proofs of this lemma and other statements can be found in Appendix A of [104].

We now establish noninterference for terminating runs.

Lemma 2 *Assuming a typable program c and ignoring diverging runs, c satisfies noninterference:*

$$\forall v_h, {v_h}', v_l \in D \ : \ \text{if } c(v_h, v_l) \neq \bot \neq c({v_h}', v_l) \quad \text{then} \quad c(v_h, v_l) = c({v_h}', v_l).$$

In particular, the above lemma tells us that (ignoring nonterminating runs), the single-run knowledge is unaffected by variation in the secret input. Thus, considering nontermination, the attacker can only observe one of two results, meaning the program only leaks one bit. The following lemma shows that this extends to the multi-run case, by showing that either termination depends only on the secret, or only on the public input. This means that the attacker can not improve their knowledge of v_h beyond the one bit already leaked, by varying the public input.

Lemma 3 *Assume c is a typable program. Then for arbitrary $v_h, {v_h}', v_l, {v_l}' \in D$ either one of the following condition holds.*

1. *Fixing the secret input, varying the public input reveals nothing:*

$$k_{v_h}(c, v_l) = k_{v_h}(c, {v_l}')$$

2. *Fixing the public input, varying the secret input reveals nothing:*

$$k_{v_h}(c, v_l) = k_{{v_h}'}(c, v_l)$$

The basic idea of the proof for this lemma, is that using Lemma 2 and assuming that neither condition holds, we can find a pair of high and low inputs that cause the program to diverge, while either of them can be combined with other inputs to cause the program to terminate successfully. By looking at the guard for the loop that causes divergence, its value must be governed by both high and low data, and so it cannot possibly have been allowed by the type system. Thus the assumption that neither condition holds must be false. As before, the full proof is presented in Appendix A of [104].

We can now use the above results to prove that typable programs leak at most one bit.

Theorem 1 *Assume c is a typable program. Then c leaks at most one bit, i.e., for all $v_h \in D$ there are at most two distinct values for K_{v_h}.*

3.5 Enforcing general knowledge policies

We now draw on ideas of delimited release [106] to change our type system so that it enforces a general knowledge policy. Delimited release specifies a declassification policy as a set of expressions called <u>escape hatches</u>. Such expressions can refer to secret variables, but their computed values may be assigned to public variables. Thus, an escape hatch defines what secret information may be declassified as public. Note that the value of an escape hatch is not released automatically, but the program can use it to compute low confidentiality information that is then released explicitly as the public output.

The knowledge policy, a partition of D, is specified with an expression e_P. In terms of delimited release, this expression is an escape hatch, and to focus on the interesting ideas of this section we assume it is the only one. Since the point of a policy expression is to partition the input space of h, any useful policy expression will only depend on h. Thus, we consider escape hatches that only involve high variables and generate policies from escape hatches as follows:

Definition 7 *An expression e, involving no other variables than h, generates a knowledge policy P as follows:*

$$P = \{P_1, \ldots, P_n\}$$

where for all v and v' we have $e(v) = e(v')$ if and only if $[\![v]\!]_P = [\![v']\!]_P$.

In order to support knowledge policies, we extend the type system with the possibility of *declassification*. The escape hatch expression is explicitly declassified to have the level LH, even though it may involve high confidentiality or low integrity variables. We adapt the definition of lev accordingly:

$$lev(e) = \begin{cases} LH & \text{if } e = e_P \text{ or } e = n \\ label(x) & \text{if } e \neq e_P \text{ and } e = x \\ lev(e_1) \sqcup lev(e_2) & \text{if } e \neq e_P \text{ and } e = e_1 \, op \, e_2 \end{cases}$$

The only typing rule that needs to be changed from Figure 7 is the one for assignment, which disallows updates to any variable involved in the escape hatch:

$$\text{ASSIGN}\ \frac{lev(e) \sqcup pc \sqsubseteq lev(x) \qquad x \notin vars(e_P)}{pc \vdash x := e}$$

This is done in order to prevent information about the secret input being laundered through the escape hatch and is standard in delimited release [106]. See Program 7 for an example of laundering.

If the high input is a tuple of multiple high inputs, as described earlier, the ASSIGN rule should simply require that x is not one of them. We have left it as is in the interest of readability.

We return to the examples of Section 3.1 to illustrate the soundness and precision of the enforcement. Program 1 is still typable independently of escape hatches. Programs 2 and 3 are rightfully rejected in the absence of escape hatches because they might leak the entire secret. Given the escape hatch $h\%4$, the secure programs 4 and 6 are accepted by the type system because declassification relabels $h\%4$ to LH, which is under LL in the lattice, the label of l. Given the same escape hatch, the insecure program 5 is rejected because $h\%6$ of type HH is assigned to variable l of type LL. Program 7 is also rejected because variable h (which is involved in an escape hatch) is modified.

The soundness of the type system is guaranteed by the following theorem.

Theorem 2 *Assume c is a typable program and e_P is an escape hatch that induces a policy P. Then c is 1-bit secure with respect to the policy P.*

3.6 Related work

Language-based information-flow security is a large and continuously-evolving field [105]. We focus on discussing most related work on knowledge-based security, interactive security, and declassification policies.

Knowledge-based security Dima et al. [51] consider sets as representation of attacker's knowledge in nondeterministic systems. Askarov and Sabelfeld [8] present a knowledge-based condition of *gradual release* for declassification, as well as enforcement for a language with communication primitives. Gradual release allows the knowledge of the attacker to increase only when the program passes a declassification point.

Van der Meyden [120] expresses intransitive noninterference policies using a classical model of knowledge in terms of different agents' views of the world [56].

Banerjee et al. [14] enhance the knowledge-based representation of attackers with powerful program specification policies. As a result, they are able to express declassification policies of both *what* can be released and *where* in the code.

Askarov and Sabelfeld [10] use knowledge to describe both termination-insensitive and -sensitive security definitions with possibilities of expressing of *what* can be released and *where*, as well as dynamic enforcement for a language with dynamic code evaluation and communication primitives.

Broberg and Sands [30] describe *paralocks*, a knowledge-based framework for expressing versatile declassification policies, including role-based policies.

Demange and Sands [47] allow tuning sensitivity to (non)termination depending on the size of the secret that is involved in loop guards: looping is disallowed when loop guards depend on secrets of small size.

None of the above approaches model the attacker's knowledge obtained by running the program multiple times.

Interactive security Multi-run security is related to interactive security. In particular, multi-run security of a batch-job program c that operates on secret variable h and public variable l can be related to single-run security of the following interactive program:

$$h' := h; \texttt{while}\ 1\ \texttt{do}\ (\texttt{in}(l); c; \texttt{out}(l); h := h')$$

where h' is an auxiliary variable. This encoding allows us for direct comparison with security definitions of interactive programs.

Le Guernic et al. [72] as well as Askarov and Sabelfeld [8] ignore diverging runs of interactive programs, which, as pointed out previously [14, 6], always allows program like c in the encoding above to be arbitrarily insecure.

ONeil et al. [94] investigate termination-sensitive security for programs that interact with input/output strategies, where strategies are represented as functions that compute the next input to the program based on the previous communication history. Being termination-sensitive and declassification-free, their condition rejects all of programs 1–7 from Section 3.1, if plugged to the encoding above.

Clark and Hunt [41] show that for deterministic programs, it makes no difference whether the user is represented by a strategy or an input/output stream. Askarov et al. [6] and Bohannon et al. [26] consider stream-based termination-insensitive security. However, as shown in [6], brute-force attacks similar to programs 2–3 are allowed.

Köpf and Basin [68] propose an information-theoretic model for multi-run security in the context of side-channel attacks. The timing side channel can be thought of as a generalization of the termination channel as nontermination manifests itself as long-lasting computation for a real-world attacker. Their model is based on refining the attacker's knowledge over multiple runs, well in line with our approach. However, as the motivation of Köpf and Basin's model is quantitative information leaks, they reason about finite numbers of runs and explore the space between our single-run and multi-run security definitions. Further, their enforcement is of rather different nature from ours: it is based on quantitative approximation using greedy heuristic.

Askarov and Sabelfeld [10] explore stream-based definitions for both termination-insensitive and -sensitive security in the presence of declassification policies. However, similar to the approaches above, the termination-sensitive condition rejects programs all of programs 1–7 and the termination-insensitive condition allows attacks 2–3, when plugged to the encoding above.

We have studied extensions of the multi-run secure type system presented here to interactive programs. Maintaining the 1-bit guarantee of termination-insensitive enforcement across all high inputs is non-trivial, as any public side effect (both input and output) will reveal information about the program counter to an attacker. If such an effect appears after a potentially diverging loop on high data, this will already leak one bit before the program has stopped. We envision that full integration of robust declassification and delimited release for interactive programs might be promising in this direction (see the discussion of dimensions of declassification below), but we expect problems with permissiveness of the enforcement. This indicates a fundamental trade-off between interactivity and security. This section identifies a niche, where it is possible to gain permissiveness without sacrificing security.

Declassification Recall that our declassification policy is an adaptation of *delimited release* [106]. Similarly to Askarov and Sabelfeld [10], we derive knowledge sets from escape-hatch expressions. The treatment of integrity by the type system is inspired by *robust declassification* [124, 91, 7]. Robust declassification guarantees that the attacker may not affect what is released by programs by ensuring that only high-integrity data can be declassified, and only in high-integrity context. In a similar spirit, our type system demands that loop context and guards may not mix high confidential data with attacker-controlled data.

In order to prevent unintended laundering of secrets, delimited release ensures that values of escape-hatch expressions do not change within a single run. In general, this guarantee does not extend over multiple runs, which potentially provides a laundering opportunity if the expression depends on data that is provided by an attacker, or is otherwise non-deterministic between runs. We avoid this issue at its root by not allowing non-secrets in escape hatches.

As we have foreshadowed earlier, we are able to cleanly reuse the robust declassification enforcement technique. However, note that we cannot automatically extract soundness guarantees from soundness results for robust declassification (e.g., [91]). The reason is that robust declassification addresses the *where* dimension of declassification: ignoring exactly *what* is leaked, but making sure the active attacker may not affect the declassification mechanism to leak more than the passive attacker. In contrast, our declassification policies are strict about *what* is leaked: the escape hatches describe the upper bound on leaks in programs.

Other, less related, work on declassification is described in a recent overview of the area [110]. The overview is organized by the dimensions of declassification.

3.7 Conclusions

We have showed how that extremes of insecurity (as with termination-insensitive noninterference) and over-restrictiveness of enforcement (as with termination-sensitive noninterference) can be avoided when generalizing batch-job security to multiple runs. Addressing the problem, we have presented a knowledge-based framework for specifying and enforcing multi-run security policies. The policy framework includes possibilities for declassification. The type-based enforcement tracks both confidentiality and integrity labels and guarantees multi-run security.

We expect interesting implications of our result for multi-threaded programs. The termination channel can be magnified in single-run multi-threaded programs in a fashion similar to using multiple runs of sequential programs. Assume we have as many threads as there are bits in secret h. Then, the multi-threaded program, where individual thread i is described as follows

$$T_i : \quad (\texttt{while}\ h\&\&b_i\ \texttt{do skip});\ \texttt{out}(i)$$

where b_i contains all zeros in the boolean representation except for bit i, leaks the entire secret in a single run. Our type-based enforcement can be straightforwardly applied to prevent this kind of leaks by considering the thread-dependent data b_i as low integrity. We expect that whenever a collection of threads is typable according to our type system, then the multi-threaded program that consists of the collection of threads is both single-run and multi-run secure (for a notion of *possibilistic* [86, 117, 109] security suitable for reasoning about nondeterministic programs).

As mentioned earlier, the termination channel can be seen as an instance of the timing side channel as nontermination manifests itself as long-lasting computation for a real-world attacker. We can offer protection against timing attacks that is similar to the protection against termination attacks: when the computation does not mix secret and attacker-controlled data in branch guards, then the timing leaks cannot be magnified. Otherwise, we resort to such existing approaches as cross-copying [3] and predictive black-box mitigation [11].

Note that there is nothing fundamental about our enforcement being static. We expect a dynamic mechanism, such as a monitor for delimited-release like policies by Askarov and Sabelfeld [10] to be easily adaptable to dynamically track both confidentiality and integrity in order to enforce our security condition.

Although this work operates on a simple two-level security lattice, we do not anticipate difficulties with extending our approach to arbitrary lattices. Requiring the confidentiality level of a loop guard to be bounded by its integrity level gives us a way to prevent the dangerous mix of high-confidentiality and low-integrity data to affect the termination behavior. Other future work focuses on expressing multi-run security for richer languages. Further, we plan to extend the framework to take into account modifications of secret data between program runs. We are also exploring decentralized security policies by knowledge-based representations of multiple attackers.

4 Reactive non-interference for the browser

4.1 Introduction: Problem statement

A browser interacts with a variety of web sites, and possibly executes JavaScript code downloaded from them. In order to make sure that these sites do not interfere in undesirable ways, today's browsers enforce the same-origin policy, an access-control policy where browser resources are tagged with their origin, and access to resources is limited to code coming from the same origin. Origin is defined as a triple (protocol, host, port), so two origins are considered to be the same only if all the elements of their tuples are equal.

The same-origin-policy has many problems, and has been criticized by many authors [115, 66]. Some of the issues, such as the fact that different browser resources use different definitions of origin, can be considered implementation bugs or inconsistencies, and they could in principle be addressed without fundamentally changing the same-origin access control policy (even though, as Singh et al. point out [115], the incompatibility burden of such fixes can be substantial). While such issues are important, they are not what we try to address here.

Other limitations of the same-origin-policy are more fundamental, and do not seem to be solvable without significant changes to the policy enforced by the browser. In particular, there are several scenarios that indicate that a policy based on non-interference would have advantages over the current access control policy. A first, very simple, motivating example is a scenario where a website sends code to perform calculations on user private data.

Example 1 (*Tax Calculator*) *Suppose the fictitious website `http://taxcalc.com` offers the service of pre-calculating the amount of tax in function of income, age, marital status and so forth. The service sends an HTML form for entering the user's information, and JavaScript code for calculating the tax based on the information entered in the form.*

The user wants assurance that the information he enters does not leave his computer — not even to the website providing the service. The same-origin-policy does not offer any protection for this scenario since the origin of the script for calculating the tax is the same as the origin of the including page.

More fundamentally, if we assume that further interactions between the user and the website are essential (for instance to pay for the service), no access-control policy can provide this assurance: the script needs access to the private data to perform its function, and it needs access to the network to send invoicing information to the service. What is needed is an information flow enforcement mechanism that can ensure that the script cannot leak private information to the network.

Example 2 (*Flight ticket*) *Consider an e-commerce site where users can order flight tickets. Obviously, the user will be fine with sharing some private information such as name, birth date and even credit-card information with the website. However, the user would like to have assurance that this information does not leak to other sites.*

The same-origin-policy provides some protection for this scenario: it ensures that scripts running in the user's browser and belonging to web pages from other origins cannot access the information entered by the user. However, scripts that are part of the e-commerce web pages will have the same origin, so they can access and easily transmit information to other sites. This can be done by initiating an HTTP request to that other site where some information to be leaked is encoded in the URL or parameters of the request [65]. The script that leaks the information does not necessarily come from the trusted site, there are many ways in which malicious scripts can find their way into pages from trusted websites. Two common attack vectors are (1) cross-site scripting (XSS), where a vulnerability in the server software enables an attacker to inject scripts in the web pages served by the server [66], and (2) the inclusion of advertisements from third-party ad-providers; such advertisements are regularly implemented as scripts that run within the same origin as the including page [119].

An important additional challenge is that for many web applications, some form of information flow between origins is actually desired. So any proposed browser security policy should not block such information flows. It is, for instance, common to include content (e.g. images and scripts) from other origins in web pages. A strict non-interference policy would prohibit such techniques and hence be strongly incompatible with the current web.

The examples above illustrate that non-interference is a promising candidate for a (baseline) browser security policy, but two important problems need to be addressed.

First, an enforcement mechanism for non-interference at the level of the browser is needed. While several browser security countermeasures based on information flow security techniques have been proposed, none of them can enforce non-interference for the full browser and for a broad class of security lattices in a secure and precise way (see the Related Work section). We propose an enforcement mechanism, and prove it secure and precise.

Second, non-interference is parameterized with a policy: a poset of security levels, and an assignment of such levels to browser inputs and outputs. Selecting suitable policies is a challenge. We analyze several interesting policies and show that they can securely handle the scenarios above, yet stay compatible with desired cross-origin information flows such as image and script loading.

4.2 Background

To address the first problem (the development of a general, secure and precise enforcement mechanism for a full browser), we need a formal model of a browser and a formalization of non-interference for such a model. This section summarizes work by Bohannan et al. on Featherweight Firefox [25] and reactive non-interference [26] that we build on in this work.

4.2.1 Reactive systems

At the highest level of abstraction, a browser is modeled as a *reactive system* [25, 26], a particular kind of automaton that reacts to inputs by changing state and emitting outputs.

Definition 8 *A reactive system is a tuple*

$$(\mathit{ConsumerState}, \mathit{ProducerState}, \mathit{Input}, \mathit{Output}, \longrightarrow)$$

where $\longrightarrow$ *is a labelled transition system whose states are* $\mathit{State} = \mathit{ConsumerState} \cup \mathit{ProducerState}$ *and whose labels are* $\mathit{Act} = \mathit{Input} \cup \mathit{Ouput}$*, subject to the constraints:*

- *for all* $C \in \mathit{ConsumerState}$*, if* $C \xrightarrow{a} Q$*, then* $a \in \mathit{Input}$ *and* $Q \in \mathit{ProducerState}$*,*
- *for all* $P \in \mathit{ProducerState}$*, if* $P \xrightarrow{a} Q$*, then* $a \in \mathit{Output}$*,*
- *for all* $C \in \mathit{ConsumerState}$ *and* $i \in \mathit{Input}$*, there exists a* $P \in \mathit{ProducerState}$ *such that* $C \xrightarrow{i} P$*, and*
- *for all* $P \in \mathit{ProducerState}$*, there exists an* $o \in \mathit{Output}$ *and* $Q \in \mathit{State}$ *such that* $P \xrightarrow{o} Q$*.*

The system is idle and is waiting for inputs in consumer states, and it emits outputs in producer states. A reactive system can only handle one input event at a time (thus correctly modeling the fact that JavaScript event handlers are single threaded). The definition allows for non-termination: it is possible that the system never returns to a consumer state. We limit our attention in this work to deterministic reactive systems.

Reactive systems transform streams of input events into streams of output events. A *stream* is defined as a coinductive interpretation of the grammar $S ::= [] \mid s :: S$, where s ranges over stream elements. A coinductive definition of the grammar defines the set of finite and infinite objects that can be built with repeated applications of the term constructors, so a stream is a finite or infinite list of elements.

Table 1: Selected user and network I/O events.

User input	`load_in_new_window`(url) `input_text`(user_window, nat, string)
User output	`window_opened` `page_loaded`(user_window, url, rendered_doc) `page_updated`(user_window, rendered_doc)
Network input	`receive`(domain, nat, cookie_updates, resp_body)
Network output	`send`(domain, request_uri, cookies, string)

We use metavariables I and O to range over streams of inputs i and outputs o, respectively. The *behavior* of a reactive system in a state Q is defined as a relation between the input streams and output streams.

Definition 9 *Coinductively define $Q(I) \Rightarrow O$ (state Q transforms the input stream I to the output stream O) by the following rules, where C and P are respectively consumer and producer states:* $C([]) \Rightarrow []$

$$\frac{C \xrightarrow{i} P \qquad P(I) \Rightarrow O}{C(i :: I) \Rightarrow O} \qquad\qquad \frac{P \xrightarrow{o} Q \qquad Q(I) \Rightarrow O}{P(I) \Rightarrow o :: O}$$

4.2.2 Featherweight Firefox

The notion of reactive system is very abstract. To analyze potential security policies, we use a browser model that concretizes the abstract states, inputs and outputs. The *Featherweight Firefox* browser model [25] does exactly that. It includes many browser features such as multiple browser windows; cookies; sending HTTP requests and receiving HTTP responses; essential HTML elements; building document node trees, and also the basic features of JavaScript. It is implemented as an executable model in OCaml [1].

Featherweight Firefox (FF) is a reactive system, with a much more detailed definition of the input and output events, and the internal state of the browser. Input events can either come from the user (loading a URL in a new window `load_in_new_window`, entering text in a text box `input_text`, etc.), or from the network (receiving an HTTP response `receive`). Output events can be to the user (web page is updated `page_updated`, window is opened `window_opened`) or to the network (sending HTTP request `send`). The FF browser model defines precisely how the browser will react to these inputs by emitting outputs. Some of input and output events are shown in Table 1.

The FF model is surprisingly rich. We will see examples including for instance the execution of event handlers implemented as scripts in an html page.

4.2.3 ID-security, or reactive non-interference

It remains to define what it means for a reactive system (and hence FF) to be non-interferent. Bohannon et al. [26] propose a notion of *ID-security*, a termination insensitive variant of non-interference. We specialize their definitions to this case.

Let us assume that a poset of security levels is given. The predicate $visible_l(s)$ models what observers of security level l can see: $visible_l(s)$ is true iff the stream element s is visible to an observer at level l. First, we define what it means for two (input or output) streams to be equivalent up to level l.

Definition 10 *Coinductively define $S \approx_l^{ID} S'$ (S is ID-similar to S' at l) with the following rules:*

$$[] \approx_l^{ID} [] \qquad \frac{visible_l(s) \qquad S \approx_l^{ID} S'}{s :: S \approx_l^{ID} s :: S'} \qquad \frac{\neg visible_l(s) \qquad S \approx_l^{ID} S'}{s :: S \approx_l^{ID} S'}$$

$$\frac{\neg visible_l(s) \qquad S \approx_l^{ID} S'}{S \approx_l^{ID} s :: S'}$$

This definition is coinductive, meaning that the property holds on the largest possible set fixed under all the rules. We can now define when a reactive system is secure in a state Q.

Definition 11 *A state Q is ID-secure or (reactive) non-interferent if, for all l, $I \approx_l^{ID} I'$ implies $O \approx_l^{ID} O'$ whenever $Q(I) \Rightarrow O$ and $Q(I') \Rightarrow O'$.*

The definitions in this section allow us to state our first goal for this work: we want to build an enforcement mechanism ensuring that FF is reactive non-interferent.

4.3 Informal overview

Our enforcement mechanism is based on a relatively new dynamic technique for achieving non-interference: secure multi-execution [50]. The core idea of this mechanism is to execute the program multiple times (one sub-execution of the program for each security level), and to ensure that (1) outputs of a given level l are only done in the sub-execution at level l, and (2) inputs at level l are only done at level l (the sub-executions above l reuse the inputs obtained by level l; sub-executions not above l are fed a default value). So the sub-execution at level

```
var a = parseInt(document
    .getElementById('a').value);
var b = parseInt(document
    .getElementById('b').value);
var sum = a + b;
document.getElementById('c').value = sum;
document.getElementById('banner')
    .src = 'http://attacker.com?t=' + sum;
```

Figure 8: JavaScript code example

l only sees inputs of levels below l and its output could not have been influenced by inputs of a higher level. Non-interference follows easily from this observation.

Devriese and Piessens [50] have worked out this mechanism for a simple sequential programming language with synchronous I/O, and have proven its security and precision. Capizzi et al. [31] have implemented it at the level of operating system processes for the case of two security levels.

The mechanism we propose adapts this technique to reactive systems, and we prove its security (weaker than what Devriese and Piessens have shown in their setting: we lose termination- and timing-sensitivity), as well as its precision (stronger than the result by Devriese and Piessens: we show precision under much weaker assumptions).

Let us explain the mechanism by means of an example. Consider again the tax calculation example from Section 4.1. The JavaScript code in Fig. 8 models the essence of this example: the user provides private inputs (two integers) in the text fields a and b, and the JavaScript code computes their sum and displays this in text field c. We can assume this JavaScript code is a part of an event handler that fires whenever the user changes the contents of a or b.

The code in Fig. 8 shows a potential attack: the script will leak the (secret) sum to "attacker.com" by sending an HTTP request to that domain with the secret as a parameter (setting the `src` property of an image HTML element in JavaScript will have as a side effect that the image is reloaded from the URL assigned to the `src` property). Recall that the JavaScript code was not necessarily endorsed by the tax calculation site. It could have been injected through a cross-site scripting (XSS) attack or hidden in an advertisement running on the page. Under a policy that assigns a high security level (H) to text field inputs, and a low security level (L) to all the outputs (including the one to "attacker.com"), this program is clearly not secure: high inputs leak to low outputs. Notice that all current browsers are vulnerable to such attack since a script gets assigned the same origin as the including page and hence is able to leak any (secret) user information.

Our enforcement mechanism runs several sub-executions of the web browser, one for each security level. Tables 2 and 3 show what happens at L and H sub-executions. The level of every input event is shown in column 1, while levels of

Table 2: Run of L sub-execution of the browser.

L		`load_in_new_window`("http://taxcalc.com")
	H	~~`window_opened`~~
	L	`send`("taxcalc.com", request_uri, cookies, ...)
L		`receive`("taxcalc.com", 0, cookie_updates, doc(a=0, b=0, c=0, js_inline))
	H	~~`page_loaded`(user_window("taxcalc.com"), ..., doc(a=0, b=0, c=0, js_inline))~~
H		~~`input_text`(user_window("taxcalc.com"), 1, "2")~~
L		(further L input)
	L	`send`("attacker.com", request_uri, cookies, "?t=0")

output events are shown in column 2. The tables show which events get suppressed. For instance, for the L sub-execution, the following events get suppressed: (1) the input events of level H (and also all output events that would have been the result of the input event), and (2) the output events at level H.

The offending output to "attacker.com" is suppressed, as the L sub-execution never gets the H input event where the user is typing secret data in the text box. In the tables, we show that even if the script would try to send the contents of a and b later in response to further L input, the actual output sent to "attacker.com" would only contain the sum of the default values in both text boxes. Notice the proposed mechanism also ensures that there are no implicit flows, since the low sub-execution (that is allowed to send) does not contain any high-level information (the user secret data). There is never *any* information flow from H inputs to L outputs.

4.4 Formalization

We propose to apply the secure multi-execution technique to a reactive system. Given an information flow policy, we build a new reactive system that is called a *wrapper*. The wrapper runs multiple *sub-executions* of the original reactive system: one for each security level. When it consumes an input event, it is passed to those sub-executions that are allowed to see it, i.e. the sub-executions at a level higher or equal than level of this event. A sub-execution produces output events only at the level of this sub-execution. Because of space constraints, proofs are provided in a separate technical report [23].

Table 3: Run of H sub-execution of the browser.

L	`load_in_new_window`("http://taxcalc.com")		
		H	`window_opened`
		L	~~`send`("taxcalc.com", request_uri, cookies, ...))~~
L	`receive`("taxcalc.com", 0, cookie_updates, doc(a=0, b=0, c=0, js_inline))		
		H	`page_loaded`(user_window("taxcalc.com"), ... doc(a=0, b=0, c=0, js_inline))
H	`input_text`(user_window("taxcalc.com"), 1, "2")		
		H	`page_updated`(user_window("taxcalc.com"), doc(a=0, b=2, c=2, ...))
		H	`window_opened`
		L	~~`send`("attacker.com", request_uri, cookies, "?t=2")~~
L	(further L input)		
		L	~~`send`("attacker.com", request_uri, cookies, "?t=2")~~

4.4.1 Secure multi-execution of reactive systems

The information flow policy contains a partially ordered set of security levels $(\mathcal{L}, \leq)$ and a function $\mathrm{lbl} : \mathit{Act} \rightarrow \mathcal{L}$ assigning security levels to all inputs and outputs of the reactive system. The output $\cdot$ is invisible at all levels, and can be used to represent internal activity of the system. (For instance to return from a producer state to a consumer state without producing real output [26].)

A state of the wrapper is a tuple (R, L), where

- R is a function mapping security levels to states of the reactive system, $R : \mathcal{L} \rightarrow \mathit{State}$. $R(l)$ is the state of the sub-execution at level l.
- L is the list of the levels of all the sub-executions that are in producer state (you can think of it as the scheduler's *ready queue*).

States $(R, \emptyset)$ are consumer states of the wrapper and states (R, L) with $L \neq \emptyset$ are producer states. The initial state of the wrapper is a state $(R, \emptyset)$ such that for all $l \in \mathcal{L}$, the state $R(l)$ is the initial state of the original reactive system.

Fig. 9 shows the semantics of the wrapper. When a new input event i arrives, it is passed to the copies at the levels in $\mathit{Upper}(i)$ (defined as a list of security levels higher or equal than the level of i), and the wrapper makes a transition to a producer state ([LOAD]). Once the wrapper is in producer state $(R, l :: L)$, it gives the sub-execution at level l a chance to proceed. If this sub-execution produces an output at level l, the wrapper outputs it ([OUT-P] and [OUT-C]), otherwise a silent

$$\text{LOAD}\ \frac{R(l) \xrightarrow{i} P_l \qquad \text{if } \mathrm{lbl}(i) \leq l \text{ then } R'(l) = P_l \qquad \text{else } R'(l) = R(l) \text{ for all } l}{(R, \emptyset) \xrightarrow{i} (R', Upper(i))}$$

$$\text{OUT-P}\ \frac{R(l) \xrightarrow{o} P \qquad \mathrm{lbl}(o) = l}{(R, l :: L) \xrightarrow{o} (R[l \mapsto P], l :: L)}$$

$$\text{OUT-C}\ \frac{R(l) \xrightarrow{o} C \qquad \mathrm{lbl}(o) = l}{(R, l :: L) \xrightarrow{o} (R[l \mapsto C], L)}$$

$$\text{DROP-P}\ \frac{R(l) \xrightarrow{o} P \qquad \mathrm{lbl}(o) \neq l}{(R, l :: L) \xrightarrow{\cdot} (R[l \mapsto P], l :: L)}$$

$$\text{DROP-C}\ \frac{R(l) \xrightarrow{o} C \qquad \mathrm{lbl}(o) \neq l}{(R, l :: L) \xrightarrow{\cdot} (R[l \mapsto C], L)}$$

Figure 9: Semantics for secure multi-execution of a reactive system.

output $(\cdot)$ is produced instead ([DROP-P] and [DROP-C]). If the sub-execution at the level l reaches a consumer state, then this level is removed from L ([OUT-C] and [DROP-C]).

It is intuitively clear that this construction guarantees non-interference. Output at level l is only produced from the sub-execution at level l, which only gets input at level l or lower, so leaks from higher levels are impossible. On the other hand, the sub-execution at level l receives identical input on level l or lower. Therefore, if the program is non-interferent, then our wrapper still produces the same output as the original. It is possible that the order of outputs will be reordered though. We will discuss both of these aspects (*security* and *precision*).

4.4.2 Security

First, we show formally that our technique guarantees termination-insensitive non-interference: for any reactive system and any information flow policy, our wrapper will never produce information leaks.

Bohannon et al. proposed a bisimulation-based proof technique based on *ID-bisimulation* relation (written $\sim_l$)[26, Definition 4.1]. Our proof is based on the key theorem of Bohannon et al.[26, Theorem 4.5] stating that if $Q \sim_l Q$ for all

l, then Q is ID-secure. In order to obtain ID-bisimulation relation on the wrapper states, we propose a definition of l-similarity.

Definition 12 *The state of the wrapper (R_1, L_1) is l-similar to the state (R_2, L_2) (written $(R_1, L_1) \approx_l (R_2, L_2)$) iff a) $R_1 \approx_l R_2$ meaning $\forall l' \leq l : R_1(l') = R_2(l')$), and b) $L_1|_l = L_2|_l$, where $L|_l$ represents the list of levels l' in L such that $l' \leq l$.*

Then we have proved the following key lemma.

Lemma 4 *The l-similarity relation is an ID-bisimulation.*

Since for every state (R, L) of the wrapper we have $(R, L) \approx_l (R, L)$, we can finally use the Theorem 4.5 from [26] and prove the security theorem.

Theorem 3 (Security) *All the states of the wrapper are ID-secure.*

4.4.3 Precision

On the other hand, we need to prove that our enforcement mechanism is precise: since it will sometimes modify the behaviour of programs, we need to prove that it does this in a sensible way, i.e. it does not observably modify behaviour for programs that already are secure. We show precise formal results to explain exactly what we mean by this.

First, we need to define what we mean by saying that our enforcement mechanism *does not observably modify the behaviour of programs*. It is important to notice that even for well-behaved programs, the wrapper can change the relative order of output events at different security levels. We assume that any observer will only observe at a single security level. This assumption is valid for the policies we will consider in Section 4.5. Then, we define the $\text{observer-indistinguishable}_l$ relation that relates input or output streams that "look the same" for observers at security level l. Like Bohannon et al., we use a coinductive definition to clearly specify this definition for infinite streams.

Definition 13 *Define* $\text{observer-indistinguishable}_l(S, S')$ *coinductively with the* *following rules:*

$$\text{observer-indistinguishable}_l([], [])$$

$$\frac{\text{lbl}(s) \neq l \qquad \text{observer-indistinguishable}_l(S, S')}{\text{observer-indistinguishable}_l(s :: S, S')}$$

$$\frac{\text{lbl}(s') \neq l \qquad \text{observer-indistinguishable}_l(S, S')}{\text{observer-indistinguishable}_l(S, s' :: S')}$$

$$\frac{\text{observer-indistinguishable}_l(S, S')}{\text{observer-indistinguishable}_l(s :: S, s :: S')}$$

This notion is weaker than Bohannon et al.'s ID-similarity. In fact, we have the following result:

Lemma 5 *If $O \approx_l^{ID} O'$, then:* observer-indistinguishable$_{l'}(O, O')$ *for all $l' \leq l$.*

Another notion we need is the projection of a finite stream at a certain security level l. The projection function π_l removes from the stream those events that are at a level not below l.

Definition 14 *Define, for finite I_0*

$$\pi_l([]) = [] \quad \pi_l(i :: I_0) = \begin{cases} \pi_l(I_0) & \textit{if } \text{lbl}(i) \not\leq l \\ i :: \pi_l(I_0) & \textit{if } \text{lbl}(i) \leq l \end{cases}$$

Our enforcement mechanism produces observably equivalent outputs for those inputs for which the original reactive system is already "well-behaved" with respect to the security policy. We use the following precise definition:

Definition 15 *Given a reactive system state Q and a finite input I and output O such that $Q(I) \Rightarrow O$ we say that Q behaves securely for input I iff for all $l \in \mathcal{L}$, we have that $Q(\pi_l(I)) \Rightarrow O_l$ with* observer-indistinguishable$_l(O, O_l)$.

These are the definitions we need to state the first of our precision theorems. The following theorem is the most detailed result, and shows that for those inputs for which the reactive system behaves securely, the corresponding wrapper produces results that are observationally equivalent.

Theorem 4 (Precision for individual runs) *Suppose a given reactive system state Q behaves securely for input I and $Q(I) \Rightarrow O_Q$. Define the corresponding wrapper $W = (R_Q, L)$ with $R_Q(l) = Q$ for all $l \in \mathcal{L}$, $L = \emptyset$ if $Q \in \mathit{ConsumerState}$ and $L = \mathcal{L}$ if $Q \in \mathit{ProducerState}$. For O_W such that $W(I) \Rightarrow O_W$, we have that $O_Q \approx^{obs} O_W$.*

This theorem is actually not a typical precision result for an information flow enforcement technique, because it does not require non-interference of the original system, as would be more typical (see e.g. Devriese and Piessens [50]). Instead, the theorem gives a sufficient condition for an individual execution to "behave securely" and produce observationally equivalent results. However, we can show that the previous theorem is stronger, by showing that if the original system was non-interferent, then all of its executions "behave securely".

Lemma 6 *If a given reactive system state Q is ID-secure, then it behaves securely for any input I.*

This lemma easily leads to the following, more classical, precision theorem.

Theorem 5 (Precision) *Suppose a given reactive system state Q is ID-secure, and $Q(I) \Rightarrow O$. Define the corresponding wrapper $W = (R_Q, L)$ with $R_Q(l) = Q$ for all $l \in \mathcal{L}$, $L = \emptyset$ if $Q \in \mathit{ConsumerState}$ and $L = \mathcal{L}$ if $Q \in \mathit{ProducerState}$. For O' such that $W(I) \Rightarrow O'$, we have that $O \approx^{obs} O'$.*

The stronger result is important in practice. Featherweight Firefox (without secure multi-execution) is never ID-secure: even if all scripts that have been loaded up to now behaved fine, somewhere in the future a malicious script might be loaded that leaks information. So the classical precision theorem does not apply, and it does not allow us to conclude precision for runs of the browser that actually behave well.

So what we need is a theorem that says: if the run of the browser up to some point behaved well, then our enforcement will not modify that run in an observable way. This is exactly what our first precision theorem does.

Note that we are only talking about precision here: security is never at stake. Featherweight Firefox with our enforcement mechanism will always be ID-secure. The point here is that we want to relate the behavior of the secured browser with the unsecured one, and this cannot be done with a classical precision theorem.

4.5 Information flow policies

We have implemented our information flow enforcement technique for Featherweight Firefox model in OCaml [1]. This implementation allows us to demonstrate

[1]It can be accessed here: http://disi.unitn.it/~bielova/sme-firefox.

valuable information flow policies for web browsers. The three basic policies we show demonstrate on the one hand the power of information flow policies, allowing us to define precisely the property that we want to enforce. On the other hand, our examples show that it is our technique that enforces the policies in a way that non-complying programs are dealt with as precisely as possible.

4.5.1 Policy 1: High/Low Policy

A first, very simple but useful policy that can be enforced classifies all user inputs as H and all network outputs as L. This is essentially the policy we used in Section 4.3 to explain our enforcement mechanism.

According to this simple High/Low policy, no public outputs are possible after secret inputs. It might seem that this will block any request to a website. But this is not the case. Intuitively, the reason why the request to “attacker.com” is being blocked is that it is made in response to a user input event, which is considered a private (H) information by our policy. Toward observers on the L security level, the policy enforcement therefore replaces this behavior by default behavior coming from the L execution, which is kept under the illusion that no user input has occurred.

Note that this policy is a very simple information flow policy, but already achieves something that previously was not possible. We can run a website making sure that certain user information is never leaked. For example, we can think of a “Keep all information in this field inside my browser” button that you can push to prevent information entered into a field from leaving your browser. The browser’s policy enforcement could then use an enforcement technique like ours to guarantee security of the information, and in many cases without affecting the further behaviour of the site.

4.5.2 Policy 2: Separating origins

The airplane tickets e-commerce site example is more typical for a general web site. In this scenario, a level of trust is assumed between the user and the company hosting the ticketing website, in order for the ticketing company to provide useful information or services. Nevertheless, the standard same-origin-policy (SOP) is not sufficient as it allows (in practice) this data to be sent anywhere.

We believe that the basic model of SOP is actually correct. When a user enters information on a website, it is typically his intent to disclose this information to the owner of that website, but not others. Likewise, information received from a website can be trusted to be sent back to this website but not to others.

A somewhat evident idea here is to use a security lattice with three types of levels: L, M(dom) for any domain *dom* and H. The L and H levels are smaller

Table 4: Origin separation policy

User input	`load_in_new_window(...)` `input_text`(user_window(dom), ...)	L M(dom)

User output	`window_opened`
	`page_loaded(...)`
	`page_updated(...)`
		H
		H
		H

Network input	`receive`(dom, ...)	M(dom)
Network output	`send`(dom, ...)	M(dom)

resp. bigger than all others and the M(...) domains are mutually incomparable. The M(dom) level is assigned to all network events originating from or going to this domain and to all user input events that contain information destined for a page on this domain. Output events going to the user are classified as H. This policy is summarized in Table 4.

Table 5 shows the execution of a prototypical airline ticketing website script under origin separation policy. A level of input event is in column 1, a level of sub-execution that receives this input is in column 2, and a level of an output is in column 3. We see that network output to "air.com" is now permitted to be influenced by information from user input.

Something interesting happens when we consider a page that tries to download a third-party script at page load time. Imagine that upon receiving an HTTP response `receive` (at level M1), the browser attempts to send the request for a third-party script (or image) at "remote.com". Our policy marks input event `receive` as information that must be revealed only to the "air.com" domain. Hence the request to the "remote.com" should not be sent.

Of course, there is a good reason why the `receive` event should be classified at this level. If we suppose the page that is received represents the third step in the airline ticket purchasing process, and contains a summary of all data previously input by the user, then this is clearly information that we want to protect and the policy is correct to not allow this info to leak to third-party sites.

One option would be to provide support for declassification. Such techniques allow higher security information to be disclosed at lower levels under certain conditions. Declassification typically requires involvement from the sandboxed code and necessarily introduces extra complexity and weakens security guarantees. In our case, we are looking for a mechanism that can be transparently applied to existing code and declassification is not the best solution.

In fact, the problem in our example is that our information flow policy is not fine-grained enough. If we want to refine the SOP retaining maximum compatibility, we need to define a policy that does a better job of formalizing the assumptions in the current web security model. In this case, the policy does not capture the implicit notion that an HTML document contains information at different confidentiality levels. If the document specifies that it requires a certain script to function then this information must be permitted to leak to the website serving the script. In the next subsection, we discuss how this is possible without disclosing the entire document.

4.5.3 Policy 3: Sub-input-event security policies

The key to solving the issue is to assign different labels to different parts of a single input event. One simple solution is to model such an input event as a number of separate input events, so that we can give each of these parts a different level. Then our enforcement mechanism and our security and precision theorems can be applied as before. An alternative, more intuitive way of thinking about this splitting of an input event (where different levels can see a different subset of the parts of the split event), is to consider security-level dependent projections that project an input event on the part of the event visible to a specific level. Space limitations keep us from discussing this policy in more detail. An extensive discussion can be found in the technical report [23]. With these final refinements, our approach realizes a substantial improvement over the standard SOP, while maintaining compatibility with typical cross-origin interactions in modern web applications.

4.6 Related Work

Since related work has already been described broadly in an earlier deliverable, we focus on the work that is most closely related to the work reported in this Section. A first very related line of work is the work by Bohannon et al. which has been discussed extensively in Section 4.2. Next, there are several other security countermeasures that have strong similarities to our approach.

The technique of secure multi-execution as proposed by Devriese and Piessens [50] is the most closely related. This new technique is proved it to be sound and precise for a simple sequential programming language with synchronous I/O. Our work extends their work to reactive systems and hence browsers. Interestingly, the formal guarantees we get are different. Whereas [50] can prove timing-sensitive non-interference, we have to settle for termination-insensitive non-interference. The main reason for this is that we are more restricted in the reordering of output events. On the other hand, we get a substantially stronger precision result. We show precision for any well-behaved run, whereas Devriese and Piessens can only prove precision for programs that are termination-sensitively non-interferent.

A similar approach was proposed by Capizzi et al. [31] where they run two executions of operating system processes for the H (secret) and L (public) security level. They limit themselves to this simple two-element poset, but they provide an actual implementation, and report on benchmarks.

In a very recent paper, Kashyap et al. [67], generalize the technique of secure multi-execution to a family of techniques that they call *the scheduling approach to non-interference*, and they analyze how the scheduling strategy used impacts the security properties offered.

4.7 Conclusion and future work

We have studied the suitability of non-interference as a replacement for the same-origin-policy in browsers. We have shown that it is possible to enforce non-interference for a browser securely and precisely for a broad class of information flow policies. In addition we have shown that, even without any support for declassification, useful information flow policies for a browser can be defined.

In many cases, we can detect that the reactive system was not non-interferent to begin with, but it is future work to investigate what can be done in these cases. A clear possibility is to inform the user that this is the case, but we could also try to apply certain heuristics to improve precision.

An important remaining challenge is the development of efficient implementation techniques for our enforcement mechanism. A naive implementation of secure multi-execution will incur a substantial performance and/or memory overhead, since multiple copies of the same program are executed. However, given that in many cases (in particular in cases where the program being executed is non-interferent) these copies will be in sync, and significant optimizations will apply.

It is also important to evaluate the impact of the proposed policies on real web sites: while the security benefits of a non-interference policy are high, there will be a price to pay. Even though we have shown by example that some level of compatibility with the current web can be maintained, it is to be expected that many detailed incompatibilities will show up, and evaluating the cost of these – and how they could be mitigated – is a key challenge for future work.

Table 5: Origin separation policy. M1 = M("air.com"), M2 = M("attacker.com").

L	load_in_new_window("http://air.com")			
		L	H	~~window_opened~~
			M1	send("air.com", request_uri, cookies, ""))
		H	H	window_opened
			M1	~~send("air.com", request_uri, cookies, ""))~~
M1	receive("air.com", 0, cookie_updates, doc(age=0, ...))			
		M1	H	~~page_loaded(user_window, "http://air.com", doc(age=0, ...))~~
		H	H	page_loaded(user_window, "http://air.com", doc(age=0, ...))
M1	input_text(user_window, 0, "25")			
		M1	H	~~page_updated(user_window, doc(age=25, ...))~~
			H	~~window_opened~~
			M1	send("air.com", request_uri, cookies, "?t=25")
			H	~~window_opened~~
			M2	~~send("attacker.com", request_uri, cookies, "?t=25")~~
		H	H	page_updated(user_window, doc(age=25, ...))
			H	window_opened
			M1	~~send("air.com", request_uri, cookies, "?t=25")~~
			H	window_opened
			M2	~~send("attacker.com", request_uri, cookies, "?t=25")~~

5 Unifying Facets of Information Integrity

5.1 Introduction

Information integrity is a vital security property in a variety of applications. However, there is clearly more than one facet to integrity. Indeed, security textbooks [98, 59] agree that it is hard to pin down the essence of integrity, and surveys [84, 112, 105] and tutorials [60] identify a range of integrity flavors.

Integrity in the area of information flow often means that trusted output is independent from untrusted input [21]. This is dual to the classical models of confidentiality [20, 71, 44, 58], where public output is required to be independent from secret input. Integrity in the area of access control [112] 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* [112, 98].

Sabelfeld and Myers [105] 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, in general, enforcement of integrity requires proving program correctness.

Seeking to clarify the area of integrity policies, Li et al. [73] suggest a classification for data integrity policies into *information-flow*, *data invariant*, and *program correctness* policies. In a similar spirit, Guttman [60] identifies *causality* and *invariance* policies as two major types of data integrity policies.

With the classification by Li et al. [73] as a point of departure, we present a general framework for the different facets of integrity that include information-flow, invariance, and correctness aspects. Furthermore, we argue that integrity via invariance is itself multi-faceted. For example, the literature (cf. [73]) features formalizations of invariance as predicate preservation (*predicate invariance*), which is not directly compatible with invariance of memory values (*value invariance*).

This section offers a unified framework for integrity policies that include all of the facets above. A key feature of the framework is generalized invariants that can represent a range of properties from program correctness to predicate and value invariance. Our formalization shows that program correctness (which was previously identified as a separate type of integrity [73]) in fact subsumes invariance-based integrity.

Figure 10 illustrates the policy set inclusion. We comment on the characteristic policy examples that correspond to points in the diagram (the formal definitions of these policies are postponed to Section 5.2). Notation x and x' denotes the values of the corresponding variable before and after program execution. An example of a value invariance policy is $x = x'$, i.e., the value of the variable stays unchanged. An example of a predicate invariance policy is $x > 0 \Rightarrow x' > 0$, i.e., if the variable is positive initially, it must stay positive at the end of execution. Value invariance is inherently about the relation of some expression before and after the execution. On the other hand, predicate invariance is inherently about preservation of some predicate on the current memory. As we explain in detail in Section 5.2, these policies are not directly compatible because (i) in order to mimic value invariance (as in $x = x'$) by predicate invariance, the final memory needs to explicitly include the initial memory, and (ii) in order to mimic predicate invariance (as in $x > 0 \Rightarrow x' > 0$) by value invariance, the predicate to be preserved needs to be encoded, if at all possible, into expression equality.

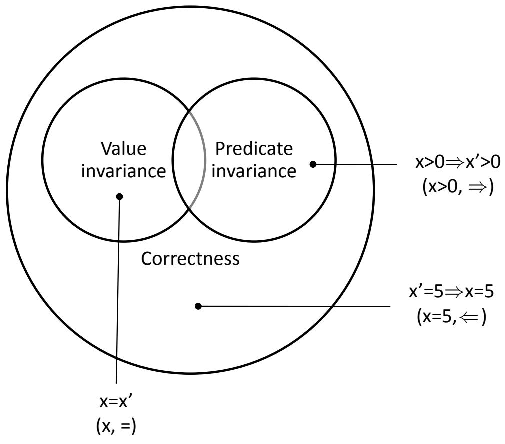

Figure 10: Generalized invariance

Further, there are properties beyond invariance that are integrity properties. For example, $x' = 5 \Rightarrow x = 5$ is a property that assures that if the final value of the variable is 5, then it has not been modified compared to its initial value. This corresponds to a general class of properties, called *program correctness* properties. Thanks to its generality, program correctness can model all of the integrity flavors, including meaningfulness and consistency. In fact, any program property can be represented as long as it can be described by a generalized predicate that has access to initial and final memories. (As we remark in Section 5.8, an extension of the framework to handle intermediate states appears natural.)

Note that the goal of this work is not to achieve as much expressiveness as possible. Indeed, a wide range of formalisms exists for reasoning about program correctness from Hoare logic [64] to refinement types [57], and a large body of work in-between [93]. Furthermore, logic-based mechanisms have been explored for reasoning about confidentiality [45, 19, 4, 14]. Instead, we aim at a treatment of integrity that allows us to express the different flavors in a uniform and convenient fashion that is directly connected to enforcement.

Indeed, despite the different nature of the integrity facets, we show that a straightforward enforcement mechanism adapted from the literature is readily available for enforcing all of the integrity facets at once. This mechanism, as proposed by Askarov and Sabelfeld [10], is originally for enforcing an information release (or *declassification*) policy of *delimited release* [106]. It guarantees that the values of declassification expressions (called *escape hatches*) have not changed compared to their initial values by performing a dynamic check at declassification time. We observe that the dual of this mechanism allows tracking both safe *endorsement* (i.e., intentional increase in trust to a given expression), and it is readily available to track correctness and therefore invariance properties. Indeed, the latter facets of integrity can be straightforwardly guaranteed by checking immediately before termination whether the desired correctness/invariance property is satisfied and terminating normally only in the case of positive outcome.

The possibility of easily deploying Askarov and Sabelfeld's enforcement [10] for a wide range of integrity policies (for which the enforcement was not originally designed) is a one of the greatest benefits of our approach. It liberates us from the necessity of designing a multi-dimensional enforcement framework of complexity similar to the policy framework.

A summary on the tightness of integration offered by our approach follows. We achieve tight integration on the enforcement side: a single enforcement mechanism is suitable to support all facets of integrity, including those that it has not been designed to support. On the policy side, the integration between information flow and correctness facets is not tight as these facets are inherently distinct. Nevertheless, within the correctness facet, we achieve tight integration of various flavors of invariants into our generalized invariant framework.

In the rest of the section, we present a generalized definition for integrity as invariance (Section 5.2), recap a standard definition of integrity as information flow (Section 5.3), show how to enforce all facets of integrity with a single enforcement mechanism (Section 5.4), discuss endorsement (Section 5.5), extensions and practical aspects (Section 5.6), related work (Section 5.7), and offer some concluding remarks (Section 5.8).

To clarify the scope of this work, we note that the focus is on *information integrity* (or data integrity), i.e., the integrity of data (in contrast to *system integrity* that addresses the integrity of the processing software and hardware units). Hence, integrity refers to information integrity throughout this section.

5.2 Integrity via invariance

Before we launch into formal definitions of the concepts described above, we need some preliminaries. In particular, we must define what it means for a program to terminate. We use the term memories for mappings from variables to values. We work with semantics given as a small-step transition system with configurations of some form $\mathcal{C}$, where the transition system defines transitions of the forms

$$\mathcal{C} \longrightarrow \mathcal{C} \quad \text{and} \quad \mathcal{C} \longrightarrow m$$

where m is a memory. A transition of the second kind represents the terminating transition. If such a transition is contained in a trace, then it will always be the last one given that there are only transitions of the above forms. An example of a configuration is the tuple $\langle c, m\rangle$ where c is a syntactic term (*command* or *program*) and m is a *memory*.

Definition 10 (Termination) *We say that configuration $\mathcal{C}_0$ terminates in a memory m, written $\mathcal{C}_0 \downarrow m$ if and only if there exists a trace*

$$\mathcal{C}_0 \rightarrow \mathcal{C}_1 \rightarrow \ldots \rightarrow \mathcal{C}_n \rightarrow m$$

(according to some particular semantics which is usually clear from the context.) If no such trace exists, we write $\mathcal{C}_0 \not\downarrow$.

Note that $\mathcal{C}_0 \not\downarrow$ covers both the cases when programs diverge, i.e., they have an infinite execution trace, or when they get stuck before reaching a terminal state.

5.2.1 Value invariance

A value invariant states that the value of an expression should not change by executing a program. We define value invariants to be expressions which are required to evaluate to the same value only in the initial and the final memory of a terminating program. We write $m(e)$ to denote the value of an expression e with respect to a memory m.

Definition 11 (Value invariant) *Let e be an expression. We say that a program c satisfies the value invariant e if and only if*

$$\forall m \,. \qquad \langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad m(e) = m'(e).$$

A simple example of a value invariant would be the expression x, corresponding to $x = x'$ in Figure 10. This value invariant states that the variable x is not modified by running the program. Note that it may be modified during the execution of the program, as long as its original value is restored in the end. Another simple example is the expression $x + y$, which allows x and y to change, as long as their changes are balanced so that their sum stays constant.

On the other hand, there are some interesting "invariants" which we cannot describe by value invariants. This includes, for example, the invariant $x > 42$, which in some ways resembles a pair of a pre- and a postcondition. Treating this boolean expression as a value invariant requires that if the expression is false in the initial memory, it must also be false in the final memory. However, by our intuition, starting from a memory where the expression is false, we would like the program to be valid no matter the final value of the expression. This leads us to another notion of invariance from the literature.

5.2.2 Predicate invariance

Predicate invariance [73] resembles very much pre- and postconditions from Hoare logic [64, 93]. A predicate invariant consists of a boolean predicate on memories that programs must preserve.

Definition 12 (Predicate invariant) *For a predicate φ on memories, a program c satisfies the predicate invariant φ if and only if*

$$\forall m\,. \qquad \langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad \varphi(m) \Rightarrow \varphi(m').$$

Predicate invariants allow us to easily describe invariants such as $x > 0$ (see Figure 10) with the intuitive semantics described above. The intuitive idea described by Li et. al. is that φ can be used to describe when a memory has a good property, where it is desirable that programs preserve that property in the final memory.

However, there are also important examples of invariants which are not captured by predicate invariance. For example, the simple value invariant x, i.e., a given variable maintains it value, cannot be modeled as a predicate invariant without passing the initial value of the variable to the final memory. Thus, the two types of invariants are incompatible. In the next section we define a new notion of invariance which unifies the two.

5.2.3 Generalized invariance

We can observe that both of the above notions of invariance quantify over all initial memories, which for deterministic languages corresponds to quantifying over all runs of a particular program. If we treat (terminating) programs purely as a transformation on memories, then a possible general notion of invariance is simply a predicate on the initial and final memories. A given program satisfies

Generalized invariance	e_1	e_2	P
Value invariance	e	e	$=$
Predicate invariance	φ	φ	$\Rightarrow$

Figure 11: Kinds of invariance

such an invariant if all pairs of initial and final memories that it relates satisfy the predicate. Obviously, this captures the two notions of invariance above.

We provide a particularly convenient policy language that is equivalent in expressiveness to a general predicate on initial and final memory. The goal is that an invariant can be easily specified by the programmer and enforced by, e.g., a runtime monitor. Thus, we specify invariants by two expressions, one to be evaluated in initial memory and one in final memory, along with a binary predicate on those values. As we will see in Section 5.5, this is a particularly beneficial way to specify invariants because of smooth integration with endorsement.

Definition 13 (Generalized invariant) *A generalized invariant is a triple* (e_1, e_2, P)*, where* e_1*, and* e_2 *are expressions, and* P *is a binary predicate on values. A program* c *satisfies such an invariant if and only if*

$$\forall m . \qquad \langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad P\big(m(e_1), m'(e_2)\big)$$

We can explore the expressiveness of this notion of invariance. We can immediately observe that it captures the previously defined notions of invariance. Any value invariant e is represented by the generalized invariant $(e, e, =)$. Similarly, any predicate invariant φ is represented by $(\varphi, \varphi, \Rightarrow)$. These observations are depicted in Figure 11.

Generalized invariants can also describe more general notions of correctness. Our example of $x' = 5 \Rightarrow x = 5$ from Section 5.1 can be described by $(x = 5, x = 5, \Leftarrow)$. If we want to make sure a certain variable increases by running a program, we can write $(x, x, <)$.

It may not be clear why we would call such a condition as the last one an invariant, as it appears to state that something has to change. However, the property $m(x) < m'(x)$ must hold for all initial and final memories m and m', if we are to say that the program in question satisfies it. In other words, the property predicate by itself is an invariant for all runs of a program.

Another important facet of integrity is that we do not want untrusted inputs to have any influence on trusted outputs. This facet cannot be described by generalized invariants [85, 113], and is the topic of the next section.

5.3 Integrity via information flow

Information-flow integrity policies restrict how untrustworthy data flows inside programs. These policies seek to prevent corrupting critical information. For example, the (untrusted) data input of an in-flight entertainment system must not affect the auto-pilot control system (critical component), but the auto-pilot control system might be allowed to display information in the in-flight entertainment systems, such as estimated time of arrival. For simplicity, we only consider two integrity levels: H_i (high integrity) for trustworthy and L_i (low integrity) for untrustworthy data. A common baseline policy for information flow is the *noninterference* policy [44, 58]. This policy states that trustworthy data cannot be affected by untrustworthy values (written as $L_i \not\sqsubseteq H_i$). However, there is no risk for untrusted data to be affected by trusted data. In this case, we indicate $H_i \sqsubseteq L_i$. The integrity levels H_i and L_i form a two-point security lattice [48] that indicates how information can flow inside programs.

As before, we write $\langle c, m \rangle \downarrow m'$ for a terminating execution of program c under the initial memory m and final memory m'. We assume that every variable in memory is assigned an integrity level. Memories m_1 and m_2 are *high-integrity equivalent*, written $m_1 =_{H_i} m_2$, if they agree on high integrity values. The following definition captures the noninterference security policy.

Definition 14 (Noninterference) *A program c satisfies noninterference if for any memories m_1 and m_2 such that $\langle c, m_1 \rangle \downarrow m_1'$, and $\langle c, m_2 \rangle \downarrow m_2'$, then*

$$m_1 =_{H_i} m_2 \quad \Longrightarrow \quad m_1' =_{H_i} m_2'.$$

The definition above ignores nonterminating executions of programs. This kind of definition is known as *termination-insensitive* noninterference [122, 105, 6]. In some cases, attackers can still affect the termination behavior of the program. However, we ignore the termination channel because its bandwidth is negligible [6] in our setting.

Information-flow integrity can be seen as the dual to confidentiality. To illustrate this connection, we assume confidentiality levels L_c and H_c for public and secret information, respectively. Observe that the integrity requirements $L_i \not\sqsubseteq H_i$ and $H_i \sqsubseteq L_i$ are the duals to the ones $L_c \sqsubseteq H_c$ and $H_c \not\sqsubseteq L_c$, which indicate that secret information cannot be leaked to public recipients. This confidentiality policy underlies the original definitions of noninterference [44, 58]. Due to this duality, the techniques developed for confidentiality can also be used to guarantee information-flow integrity. In the next section, we extend a runtime monitor for enforcing information-flow confidentiality to enforce both information-flow and invariance integrity.

$$
\begin{aligned}
&n \in \mathbb{Z}, \quad x \in \mathit{Vars}, \quad op \in \{+, -, \dots\} \\
&e ::= n \mid x \mid e\, op\, e \\
&c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c
\end{aligned}
$$

Figure 12: Syntax

5.4 Enforcement

To illustrate the idea behind enforcement, we consider a simple imperative language with the syntax and semantics given in Figures 12 and 13, respectively. The syntax and semantic rules are mostly standard [123] except for minor extensions. We include a pseudo-term end that indicates leaving the scope of an `if` or a `while`. This term generates a transition described by the rule END. The rule TERM$_c$ is also nonstandard and, together with the empty term ε, it guarantees that a terminating run of any program ends with a transition generated by this rule. Transitions in the semantics are labeled with an event β. The purpose of labeled events as well as rules END and TERM$_c$ is communication with the runtime monitor, which is described next.

We present an extension to the dynamic monitor found in [10] in order to enforce both information-flow integrity and generalized invariants.

Figure 14 gives the monitor semantics. The monitor is a separate transition system whose transitions are labeled with the same kind of events β as the command transitions. This is used to synchronize the two executions. Furthermore, the monitor may block progress of the program, in case the program can do a transition with a certain event but the monitor is not able to match it.

The monitor enforces information-flow integrity with the rules FLOW, BRANCH and FINISH, in the same way as [10]. The first rule allows direct assignments of an expression e to a variable x, indicated by the event $a(x, e)$, only if e has the same or higher integrity than x (Γ maps variables to their integrity levels.) The rule also ensures that the minimum level $lev(st)$ on the context stack st is at least as high as x's level. This is to prevent *implicit flows* [49], i.e., flows via control flow. The stack contains the levels of expressions affecting control flow. It is maintained by the rules BRANCH and FINISH, which synchronize with the program entering or leaving an `if` or `while` block, as indicated by the events $b(e)$ and f, respectively.

The rule TERM$_m$ synchronizes with program termination and enforces the invariance integrity policy. The monitor carries in its state a set of generalized invariants, as well as a snapshot of the initial memory, and uses these to ensure all the invariants are satisfied by the execution. If not, this rule blocks the program from terminating.

SKIP
$$\langle \texttt{skip}, m\rangle \xrightarrow{nop} \langle \varepsilon, m\rangle$$

ASSIGN
$$\langle x := e, m\rangle \xrightarrow{a(x,e)} \langle \varepsilon, m[x \mapsto m(e)]\rangle$$

$$\text{SEQ}_1\ \frac{\langle c_1, m\rangle \xrightarrow{\beta} \langle c_1', m'\rangle \qquad c_1' \neq \varepsilon}{\langle c_1; c_2, m\rangle \xrightarrow{\beta} \langle c_1'; c_2, m'\rangle} \qquad \text{SEQ}_2\ \frac{\langle c_1, m\rangle \xrightarrow{\beta} \langle \varepsilon, m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\beta} \langle c_2, m'\rangle}$$

$$\text{IF}_1\ \frac{m(e) \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{b(e)} \langle c_1; end, m\rangle}$$

$$\text{IF}_2\ \frac{m(e) = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{b(e)} \langle c_2; end, m\rangle}$$

WHILE$_1$
$$\frac{m(e) \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \xrightarrow{b(e)} \langle c; end; \texttt{while}\ e\ \texttt{do}\ c, m\rangle}$$

WHILE$_2$
$$\frac{m(e) = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \xrightarrow{b(e)} \langle end, m\rangle}$$

END
$$\langle end, m\rangle \xrightarrow{f} \langle \varepsilon, m\rangle$$

TERM$_c$
$$\langle \varepsilon, m\rangle \xrightarrow{term(m)} m$$

Figure 13: Command semantics

Before proving the desired properties of our monitor we should make a small note about its practicality. While it is certainly infeasible to store a snapshot of the initial memory of a program, this is only a feature of our theoretical model. In practice the only additional state required to enforce a set of invariants $\mathcal{I}$, are the values of the first expression of each one, as evaluated in initial memory. A monitor needs only evaluate these expressions at the start, store their values and then at the end evaluate the second expression as well as the predicate of each invariant. Since we expect the set of invariants to be relatively small given their expressiveness, the overhead of adding invariant enforcement is small compared to the information flow enforcement overhead of the original monitor from [10]. Controlling the complexity of the expressions and predicates of course remains the responsibility of the policy writer.

In the rest of the section, we will talk about monitored programs which refers to a program which is run in lockstep with a monitor. For convenience, we repre-

$$\text{NOP}\ \frac{}{\langle i, st, \mathcal{I}\rangle \xrightarrow{nop} \langle i, st, \mathcal{I}\rangle}$$

$$\text{FLOW}\ \frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{\langle i, st, \mathcal{I}\rangle \xrightarrow{a(x,e)} \langle i, st, \mathcal{I}\rangle}$$

$$\text{BRANCH}\ \frac{}{\langle i, st, \mathcal{I}\rangle \xrightarrow{b(e)} \langle i, lev(e) : st, \mathcal{I}\rangle}$$

$$\text{FINISH}\ \frac{}{\langle i, hd : st, \mathcal{I}\rangle \xrightarrow{f} \langle i, st, \mathcal{I}\rangle}$$

$$\text{TERM}_m\ \frac{\forall (e_1, e_2, P) \in \mathcal{I} : P(i(e_1), m(e_2))}{\langle i, st, \mathcal{I}\rangle \xrightarrow{term(m)} \langle i, st, \mathcal{I}\rangle}$$

Figure 14: Monitor semantics

sent monitored programs with a monitor combination operator $\sharp$, whose semantics is defined with the following two rules, where $\mathcal{C}_c$ is the configuration of program semantics, and $\mathcal{C}_m$ is that of the monitor semantics.

$$\frac{\mathcal{C}_c \xrightarrow{\beta} \mathcal{C}'_c \qquad \mathcal{C}_m \xrightarrow{\beta} \mathcal{C}'_m}{\mathcal{C}_c \sharp \mathcal{C}_m \longrightarrow \mathcal{C}'_c \sharp \mathcal{C}'_m} \qquad \frac{\mathcal{C}_c \xrightarrow{\beta} m \qquad \mathcal{C}_m \xrightarrow{\beta} \mathcal{C}'_m}{\mathcal{C}_c \sharp \mathcal{C}_m \longrightarrow m} \tag{8}$$

Note that $\sharp$ is a meta-operator, it works on configurations rather than syntactic terms.

We can immediately state and prove one useful property of such monitored processes. If an unmonitored program does not terminate, then adding a monitor can not make it terminate. This is obvious from the right rule above, a terminating transition of the unmonitored program is a premise for proving a terminating transition of the monitored one. Nevertheless it is useful to state this explicitly as a lemma.

Lemma 1 (Failstop correctness) *For any monitored program $\mathcal{C}_c \sharp \mathcal{C}_m$, we have*

$$\mathcal{C}_c \sharp \mathcal{C}_m \downarrow m \quad \Longrightarrow \quad \mathcal{C}_c \downarrow m.$$

Proof. If the monitored program terminates, there is a terminating trace with transitions proved by the rules (8). By taking the left premise of each transition proof, it is straight-forward to construct a terminating trace for the unmonitored program $\langle c, m\rangle$.

Since the terminating transition must be due to the right rule, it is obvious that both traces terminate in the same memory. □

Throughout this work, we assume finite sets of generalized invariants $\mathcal{I}$. Given a set of generalized invariants $\mathcal{I}$, we can now prove that the monitor presented in Figure 14 is sound, in the way that a monitored program that terminates satisfies all invariants of $\mathcal{I}$ and satisfies noninterference. An important ingredient to this is that all invariants are decidable. We assume evaluation of their expressions is decidable, but we also require that checking each invariant's predicate is decidable as well.

Theorem 2 (Soundness) *Let c be a command and $\mathcal{I}$ a set of (generalized) invariants with decidable predicates. Then, for all memories m it holds that*

$$\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \downarrow m' \quad\Longrightarrow\quad \forall(e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)),$$

i.e., the monitored program satisfies all of the invariants in $\mathcal{I}$. Furthermore, if m_1 and m_2 are high-integrity equivalent memories, and $\langle c, m_i\rangle \sharp \langle m_i, [], \mathcal{I}\rangle \downarrow m_i'$, with $i \in \{1, 2\}$, then m_1' is high-integrity equivalent to m_2', i.e., the monitored program satisfies noninterference.

Proof. We note that $\mathcal{I}$ stays unchanged by the monitor. Since $\langle c, m\rangle$ terminates in m', there exists a trace

$$\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \longrightarrow \cdots \longrightarrow \mathcal{C}_c' \sharp \mathcal{C}_m' \longrightarrow m'$$

The last rule of a monitored execution can only be the right rule of (8), which in turn means the rule used to prove the left premise is TERM_c and that $\mathcal{C}_c' = \langle \varepsilon, m'\rangle$. Consequently, the last transition of this trace must have the following proof tree:

$$\dfrac{\mathrm{TERM}_c\ \dfrac{}{\langle \varepsilon, m'\rangle \xrightarrow{term(m')} m'} \qquad \mathrm{TERM}_m\ \dfrac{\forall(e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2))}{\langle m, st, \mathcal{I}\rangle \xrightarrow{term(m')} \langle m, st, \mathcal{I}\rangle}}{\langle \varepsilon, m'\rangle \sharp \langle m, st, \mathcal{I}\rangle \longrightarrow m'}$$

The only premise in this proof must thus hold, which concludes the proof of the invariance part.

For proof of the noninterference part we refer to [10]. □

It is a natural question to ask also if the monitor is complete. Informally, we would formulate this in the following way: If a program satisfies a set of invariants to begin with, a monitored version will not diverge unless the program does also. The presented monitor enforces both information flow integrity as well as invariant integrity. The monitor is not complete in enforcing noninterference. For example, the program $h := l; h := 0$, where h and l are high- and low-integrity variables, respectively, is blocked by the monitor although it satisfies noninterference. However, we can prove that if the information-flow integrity is set aside, then the monitor is complete in enforcing invariance integrity.

We will use the following fact (that is straightforward to prove): If all variables used in a program have the same integrity level, then no execution of the monitored version $\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle$ (where $\mathcal{I}$ is arbitrary) will get stuck due to the rule FLOW being disabled. This is obvious since the premises of the rule are always true if all integrity levels are equal. We can now state and prove the completeness of the monitor with respect to invariant integrity policies.

Theorem 3 (Completeness of invariance enforcement) *Let c be a command, m some memory, and $\mathcal{I}$ a set of generalized invariants with decidable predicates. Assume all variables used in c have the same integrity level. Then, if the (unmonitored) program $\langle c, m\rangle$ satisfies the invariants in $\mathcal{I}$, i.e.,*

$$\langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad \forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)),$$

then the program either diverges by itself or the monitored version also terminates (in some memory):

$$\langle c, m\rangle \not\downarrow \quad \vee \quad \langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \downarrow m''.$$

Proof. If the premise holds because $\langle c, m\rangle$ does not terminate, then the conclusion holds trivially. In the other case, when $\langle c, m\rangle \downarrow m'$ and

$$\forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)), \tag{9}$$

then we consider the terminating trace

$$\langle c, m\rangle = \langle c_0, m_0\rangle \longrightarrow \ldots \longrightarrow \langle c_n, m_n\rangle \longrightarrow m'. \tag{10}$$

From the command semantics we can see that the last transition is due to rule TERM$_c$.

Now consider the monitored version $\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle$. If this does not terminate, it must be because the monitor blocks the execution at some point. This can only happen if rules FLOW or TERM$_m$ are disabled. However, the rule FLOW is never disabled since there is no violation of information-flow integrity, and so the monitor can only block due to the termination rule being disabled. This would mean that the monitored program gets stuck just before the last transition of (10), since this is the only transition that can potentially synchronize with TERM$_m$. This means $m_n = m'$ and thus we are already in final memory at this point. Since TERM$_m$ is disabled, its premise is false. However being in final memory, its premise is exactly (9), which we assumed true. Thus, the monitored program must terminate. □

The completeness theorem states that our monitor will never stop an otherwise terminating and correct program. In other words, the monitor does not raise false alarms.

However, completeness alone is not enough, since the monitor could potentially terminate in a different final memory than the original, correct program does. Of course, this is not desirable, so we follow with a proof that our monitor is transparent, i.e., it does not alter the semantics of correct programs.

Theorem 4 (Transparency of invariance enforcement) *Let c be a command, m a memory, and $\mathcal{I}$ a set of generalized invariants with decidable predicates. We assume that all variables in c have the same integrity level. If the (unmonitored) program satisfies the invariants in $\mathcal{I}$, formally*

$$\langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad \forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)),$$

then, the following implications hold:

$$\begin{array}{lcll} \langle c, m\rangle \downarrow m' & \Longrightarrow & \langle c, m\rangle \,\sharp\, \langle m, [], \mathcal{I}\rangle \downarrow m', & \textit{and} \\ \langle c, m\rangle \not\downarrow & \Longrightarrow & \langle c, m\rangle \,\sharp\, \langle m, [], \mathcal{I}\rangle \not\downarrow & \end{array}$$

Proof. First, assume that $\langle c, m\rangle \downarrow m'$. By the completeness theorem above, the monitored version terminates in some memory m''. To see that $m' = m''$, observe the last transition of the monitored trace. This transition is due to the right rule of (8) whose first premise can only be met by the last transition of (10) from the last proof. By the definition of that rule, the conclusion indeed "returns" the same memory m'. This proves the first implication. The second implication is a simple contrapositive of Lemma 1. □

5.5 Endorsement

When dealing with confidentiality, it is sometimes necessary to intentionally release, or *declassify*, some confidential information [110]. Analogously for integrity, it is sometimes necessary to boost the integrity of some piece of untrustworthy data to trustworthy. For example, the integrity of user-provided data can be raised after the data is sanitized. Dually to declassification, *endorsement* converts low integrity into high integrity data.

This section introduces a security condition and an enforcement mechanism for endorsement that can be seen as the dual of *delimited release* [106, 10]. We include the command $x := \texttt{endorse}(e)$ in our language for boosting the integrity of expression e from low to high. The semantic rule, depicted in Figure 15, simply performs the assignment and triggers the event $end(x, e, m)$ for communication with the monitor.

$$\frac{}{\langle x := \texttt{endorse}(e), m\rangle \xrightarrow{end(x,e,m)} \langle \varepsilon, m[x \mapsto m(e)]\rangle}$$

$$\frac{i(e) = m(e) \quad lev(st) \sqsubseteq \Gamma(x)}{\langle i, st, \mathcal{I}\rangle \xrightarrow{end(x,e,m)} \langle i, st, \mathcal{I}\rangle}$$

Figure 15: Rules for endorsement

The security condition, dubbed *delimited endorsement*, captures what it means to be secure for programs involving endorsements.

Definition 15 (Delimited endorsement) *Consider a program c containing exactly n endorsement commands $x_1 := \texttt{endorse}(e_1), \ldots, x_n := \texttt{endorse}(e_n)$, where expressions $e_1, \ldots, e_n$ are called escape hatches. Command c is secure if for all memories m_1 and m_2 such that $m_1 =_{H_i} m_2$, $\forall i. m_1(e_i) =_{H_i} m_2(e_i)$, $\langle c, m_1\rangle \Downarrow m_1'$, and $\langle c, m_2\rangle \Downarrow m_2'$, we have $m_1' =_{H_i} m_2'$.*

Intuitively, delimited endorsement establishes that a program is secure if whenever two high-integrity equivalent memories are indistinguishable by escape hatches, then they must also be indistinguishable by the program itself: terminating runs of the program in these memories leads to high-integrity equivalent final states. One way to enforce this condition is by checking whether the value of any escape-hatch expression at the time of endorsement is the same as it was at the beginning of computation. This brings us to the enforcement.

The monitor rule for endorsement is also given in Figure 15. It checks that the endorsed value $m(e)$ of expression e in memory m is indeed the same in the initial and current memory ($i(e) = m(e)$). This restriction avoids laundering, i.e., abusing the endorsement mechanisms to endorse other data than the one indicated by $x := \texttt{endorse}(e)$ [106, 10]. Similar as for regular assignments, restriction $lev(st) \sqsubseteq x$ is used to avoid implicit flows.

The mechanisms to enforce invariants in Figure 14 can be easily reused for enforcing endorsement. Observe that $i(e) = m(e)$ can be interpreted as a particular kind of invariant $P(i(e), m(e'))$, where P is the equality predicate $=$, and expressions e and e' are the same.

The following theorem establishes the formal guarantees obtained by the enforcement rules.

Theorem 5 *For any program c, the monitored execution of c (with the initial configuration $\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle$ for memory m) satisfies delimited endorsement.*

Proof. It follows by an adaptation of Askarov and Sabelfeld's proof [10] for delimited release. The failstop property of the monitor allows for a straightforward adaption of the proof: the invariant-checking part is largely orthogonal since all the monitor can do is to block the execution, in which case the high-integrity equivalence does not need to be tracked. □

As it was for the information-flow part of the monitor in Section 5.4, the delimited endorsement monitor is incomplete in the information-flow part for the same reason.

5.6 Extensions and practical aspects

The enforcement mechanisms presented in Section 5.4 and 5.5 can be naturally extended to support I/O operations and a form of access control. We briefly outline the principles behind such extensions and discuss practical aspects.

5.6.1 I/O

Programs often require to take inputs as well as produce outputs during execution. Defining and tracking delimited release in the presence of communication primitives is described in [10]. When considering inputs, the restriction $i(e) = m(e)$ needs to be revised because it does not allow to declassify (endorse in our case) variables that have been updated by inputs.

Askarov and Sabelfeld [10] remark that inputs may introduce fresh data into programs and, therefore, they distinguish them from regular updates. They propose a monitor that allows to declassify information when the value being declassified $(m(e))$ matches the value of the expression in a memory that records most recent inputs. If no inputs where performed for a given variable e, the value considered for that variable is the one found in the initial memory. In a similar fashion, it is possible to modularly extend the rules in Figures 14 and 15 to consider a *context level input* label *ct*, which records if there has been an input in a high context, and update memory i in the monitor's state every time that an input is produced. The extended monitor then disallows endorsement if the input context label ct has low integrity. This is necessary because inputs, unlike branch/loop guards, are not lexically-bounded in their impact. The update of memory i on every input allows the monitor to have a memory where each variable's value refers either to its last input or its value at the initial memory (i.e., no inputs are performed for that variable).

In the presence of outputs, checking invariants at the end of program execution needs to be revised. Data invariants could refer to outputs produced by programs, e.g., every credit-card number sent to a server must be formed by 16 digits. To express this, it is sufficient to apply rule TERM_c at every output produced by the program. In principle, it is possible to allow programmers to indicate what invariants must be checked at what outputs.

When considering inputs and outputs, the security condition for declassification in [10] is based on the attacker's knowledge [51, 8, 14]. With this in mind, it is possible to use the same semantics techniques to handle endorsement in presence of communication primitives. In fact, the dual of the attacker knowledge in [10] can be interpreted as the attacker capabilities to control or affect computations regarding high integrity data [7].

5.6.2 Access control

As mentioned in Section 5.1, integrity in the area of access control [112] focuses on preventing data modification operations when no modification access is granted to a given principal. Policies of the kind "resource R cannot be written by principal P" cannot be naturally enforced by noninterference. The main reason is the degree of freedom that noninterference allows regarding entities at the same security level. Noninterference only restricts how information flows among different security levels. To illustrate this, assume an information-flow enforcement mechanism is in place. Whatever security level variable R is assigned to, it is still possible to read its content, concatenated with itself, and save it back to R. Observe that these operations only manipulate data at security level R. In contrast, our monitor can be easily adapted to enforce that no write operation is invoked on R by P or, more generally, no changes are performed on resource R by just establishing, through an invariant, that the content of R is the same at the beginning and at the end of the program. Moreover, if considering endorsement as given in Figure 15, it is possible to enforce no changes on R by endorsing it at the end of the program. Direct enforcement of no unauthorized write operations is of course also possible when the monitor has access to the entire trace.

5.6.3 Practical aspects

Preliminary results from a Haskell-based library for integrity [52] suggest light implementation overhead to enforce integrity policies in presence of I/O and access control requirements. Diserholt [52] shows how to build a secure password administrator that preserves confidentiality of passwords as well as several facets of integrity policies, e.g., password must be difficult to guess (integrity via invariance), certain operations should not write the contents of some files (access control), and user input cannot determine the utilized hash function (integrity via information flow). We argue that it is not difficult to reformulate the concrete case study in [52] using our approach and obtain similar results.

5.7 Related work

As one of the most fundamental security properties, integrity is subject to a vast area of research. We refer to security textbooks [98, 59] that discuss assorted flavors of integrity, and integrity surveys [84, 112] and tutorials [60] that develop integrity classifications. Section 5.1 also contains pointers to various interpretations of integrity in various disciplines.

To the best of our knowledge, our framework is the first to unify information integrity for programs. As mentioned previously, our departure point is the classification by Li et al. [73]. Our contribution compared to this classification is a more general model of invariants (Li et al. only discuss predicate invariants), a more general model of information flow (Li et al. do not consider endorsement), and a unified view, where we show that program correctness subsumes invariance policies. In addition, we also offer a unified enforcement mechanism that guarantees all aspects of integrity at once.

Information-flow integrity dates back to Biba's integrity model [21], which dualizes Bell and LaPadula's model [20, 71] for mandatory access control. The Clark-Wilson integrity model [42] is a classical model that focuses on separation of duties and transactions.

Although information integrity for programs has been unexplored compared to confidentiality, it has recently received increasing attention. Languages such as Perl, PHP, and Ruby offer dynamic integrity checks that are based on *tainting*, a runtime mechanism for tracking explicit flows.

Ørbæk and Palsberg [96, 97] define instrumented information-flow semantics for integrity in λ-calculus. The semantics is based on integrity label manipulation. An unsoundness related to the impact of flow sensitivity on information flow has been recently uncovered [101].

Heintze and Riecke [63] consider integrity as dual to confidentiality in their study of information flow for a language based on λ-calculus. Li and Zdancewic unify confidentiality and integrity policies [74] in the context of information downgrading.

A line of work on robust declassification [124, 91, 7] is based on an interplay between confidentiality and integrity, where information release (of high confidentiality data) is allowed only if it cannot be manipulated by the attacker (through attacker-controlled low integrity data) to release additional information. The Java-based Jif tool [92], as well as its web-based extensions [39, 37], implement robustness policies.

Sabelfeld and Sands [110] introduce dimensions of declassification, with the main focus on declassification of confidentiality levels. They informally discuss the dual of dimensions of declassification: dimensions of endorsement.

We draw on delimited release [106] when it comes to enforcement of integrity policies. Although delimited release is a confidentiality property, its enforcement includes information-flow aspects and is capable of enforcing generalized invariants. This work builds on a runtime mechanism for delimited release by Askarov and Sabelfeld [10]. A static alternative to tracking delimited release has been explored by Sabelfeld and Myers [106].

Boudol and Kolundzija [27] combine programming constructs for expressing access-control and declassification policies. Access control is represented at language level, with explicit granting, restricting, and testing access rights. Information-flow policies and access control have been also integrated at language level by Banerjee and Naumann [15], although without considering declassification.

Haack et al. [61] explore reasoning about explicit flows in program logic. They arrive at two kinds of integrity notions: *flow-based* and *format-based*. The former is an information-flow policy, and the latter is concerned with proper formatting (they give an example policy such as “a phone number field should only contain numbers”). This latter type of integrity is subsumed by generalized invariance.

Cheney et al. [34] investigate semantic foundations for *data provenance* in databases. Provenance is concerned with tracking the origin of information, and so Cheney et al. model it as a dependency analysis.

Diserholt [52] proposes a library that handles confidentiality and integrity policies in Haskell. Besides handling confidentiality, the library is also able to combine information-flow integrity, predicate invariants, and some means for access control. Similarly to this work, they are inspired by the classification of integrity policies in [73].

Clarkson and Schneider [43] propose contamination and suppression as quantitative definitions of integrity. The former is dual to quantitative information leakage, whereas the later measures how much information is lost from outputs. The study of suppression includes program suppression due to malicious influence and implementation errors as well as channel suppression due to information loss about inputs to a noisy channel.

5.8 Conclusions

We have presented a uniform framework for information integrity. The framework incorporates a range of integrity aspects from information-flow integrity to program correctness. The framework integrates different types of integrity as invariance. We show that some of the invariant-based policies are not compatible with each other (cf. value vs. predicate invariance). Nevertheless, they are naturally represented in our framework as program correctness properties. Endorsement policies naturally extend information-flow policies and also fit into the framework.

Despite being general, our integrity framework is realizable. A single enforcement mechanism [10] (for tracking delimited information release) turns out to be an excellent match for enforcement of integrity. It supports both information-flow integrity, including extensions with endorsement policies, as well as correctness properties, including the various flavors of invariance. This mechanism is scalable to handling communication primitives.

Future work is focused on the directions outlined in Section 5.6. We explore both formal aspects of policies in the presence of communication and access control and practical aspects of enforcement, with inlining transformation and library-based enforcement as our main goals. Another direction of work is an extension of the framework to represent trace properties, i.e., properties of sequences of intermediate states. We expect the extension of the framework and monitor rather straightforward: generalized invariants can just as well refer to the full traces, and enforcement corresponds to enforcing *safety* [113] properties.

It is important to support our results with practical findings from case studies. Preliminary results from a Haskell-based library for integrity [52] suggest light implementation overhead.

Acknowledgments Thanks are due to Michael Clarkson for useful comments. This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR. Arnar Birgisson is a recipient of the Google Europe Fellowship in Computer Security, and this research is supported in part by this Google Fellowship.

6 Conclusion

Web applications are different from many other application domains in that they are frequently built up by collaborating, but mutually distrusting components.

Section 2 presents a framework for reasoning about 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.

Different attack scenarios demand different security policies. For instance, the standard program-centric attacker model is not necessarily suitable for Web applications. In addition, for practical information flow security, there is a tradeoff between the the expressiveness of the security policy and its enforceability.

With respect to this, Section 3 shows how that extremes of insecurity (as with termination-insensitive noninterference) and over-restrictiveness of enforcement (as with termination-sensitive noninterference) can be avoided when generalizing batch-job security to multiple runs. Addressing the problem, we have presented a knowledge-based framework for specifying and enforcing multi-run security policies. The policy framework includes possibilities for declassification. The type-based enforcement tracks both confidentiality and integrity labels and guarantees multi-run security.

On the enforcement side, Section 4 explores an enforcement technique for reactive non-interference based on secure multi-execution [50]. The value of the technique is shown by implementing it for Featherweight Firefox. To the best of our knowledge, this proposal is the first to enforce a general non-interference policy for the browser as a whole.

Another property of the web that makes it stand out from the standard program-centric view of security is that web applications frequently are built from a number of collaborating, but mutually distrusting components. A typical example is the mashup. This leads to the need for the creation of rich decentralized information-flow policies.

As Section 3 shows, the notion of integrity is paramount for secure information flow. In this case the notion of integrity is dual to confidentiality. However, the we argue that there are additional facets of integrity that must be considered.

To this end, Section 5 presents a uniform framework for information integrity. The framework incorporates a range of integrity aspects from information-flow integrity to program correctness. The framework integrates different types of integrity as invariance. We show that some of the invariant-based policies are not compatible with each other (cf. value vs. predicate invariance). Nevertheless, they are naturally represented in the framework as program correctness properties. Endorsement policies naturally extend information-flow policies and also fit into the framework.

Despite being general, the integrity framework is realizable. A single enforcement mechanism [10] (for tracking delimited information release) turns out to be an excellent match for enforcement of integrity. It supports both information-flow integrity, including extensions with endorsement policies, as well as correctness properties, including the various flavors of invariance. This mechanism is scalable to handling communication primitives.

As discussed, the choice of security policy does not only depend on the attacker model. In addition, the question of enforceability is important for practical information flow security. There are a number of different choices guiding the design of information flow enforcement for web applications. A major choice is whether to use a static, a dynamic, or a hybrid method of enforcement. Our work offers extensive coverage of these alternatives. Section 3 provides static enforcement in the form of a type system, whereas Section 5 develops dynamic enforcement in the form of a runtime monitor. Section 2 features a hybrid method where a dynamic monitor is inlined into the code using static analysis and transformation. Further, Section 4 features a language-independent approach that achieves secure information flow by multiple runs of a given program at different security levels.

These alternatives provide an excellent base for exploring whether the enforcement should be placed on client side, server side or as a collaboration between the two. This work is planned for the rest of the activities in the information-flow work package, to be reported in Deliverable 3.3.

References

[1] Ocaml. Software release. http://caml.inria.fr.

[2] 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, Jan. 1999.

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

[4] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In Proc. ACM Symp. on Principles of Programming Languages, pages 91–102, 2006.

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

[6] 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, Oct. 2008.

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

[8] 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.

[9] 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.

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

[11] A. Askarov, D. Zhang, and A. Myers. Predictive black-box mitigation of timing channels. In ACM Conference on Computer and Communications Security, pages 297–307, 2010.

[12] 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.

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

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

[15] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. Journal of Functional Programming, 15(2):131–177, Mar. 2005.

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

[17] A. Barth, C. Jackson, and J. 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.

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

[19] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In Proc. IEEE Computer Security Foundations Workshop, pages 100–114, June 2004.

[20] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

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

[22] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for the browser: extended version. CW Reports CW602, Department of Computer Science, K.U.Leuven, Feb. 2011.

[23] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for the browser: extended version. Technical Report CW602, CS Dept., K.U.Leuven, February 2011.

[24] 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.

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

[26] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In ACM Conference on Computer and Communications Security, pages 79–90, Nov. 2009.

[27] G. Boudol and M. Kolundzija. Access-control and declassification. In Proc. Mathematical Methods, Models, and Architectures for Computer Networks Security, volume 1 of Communications in Computer and Information Science, pages 85–98. Springer-Verlag, Sept. 2007.

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

[29] N. Broberg and D. 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.

[30] N. Broberg and D. 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.

[31] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. Sistla. Preventing information leaks through shadow executions. In ACSAC, 2008.

[32] 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.

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

[34] J. Cheney, A. Ahmed, and U. Acar. Provenance as dependency analysis. In Proc. Database Programming Languages, 2007.

[35] W. Cheng. Information Flow for Secure Distributed Applications. PhD thesis, Massachusetts Institute of Technology, Sept. 2009.

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

[37] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In Proc. ACM Symp. on Operating System Principles, pages 31–44, Oct. 2007.

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

[39] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In Proc. USENIX Security Symposium, pages 1–16, Aug. 2007.

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

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

[42] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In Proc. IEEE Symp. on Security and Privacy, pages 184–193, May 1987.

[43] M. Clarkson and F. B. Schneider. Quantification of integrity. In Proc. IEEE Computer Security Foundations Symposium, July 2010.

[44] 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.

[45] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In Proc. Workshop on Issues in the Theory of Security, Apr. 2003.

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

[47] D. Demange and D. 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.

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

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

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

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

[52] A. Diserholt. Providing integrity policies as a library in Haskell. Master Thesis, Chalmers University of Technology, Gothenburg, Mar. 2010.

[53] D. Dolev and A. Yao. On the security of public-key protocols. IEEE Transactions on Information Theory, 2(29):198–208, Aug. 1983.

[54] 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), Oct. 2005.

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

[56] R. Fagin, J.Y.Halpern, Y.Moses, and M. Vardi. Reasoning About Knowledge. MIT Press, 1995.

[57] T. Freeman and F. Pfenning. Refinement types for ml. In Proc. ACM SIGPLAN Conference on Programming language Design and Implementation, pages 268–277, 1991.

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

[59] D. Gollmann. Computer Security (2nd Edition). Wiley, 2006.

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

[61] C. Haack, E. Poll, and A. Schubert. Explicit information flow properties in JML. In Proc. WISSEC, 2008.

[62] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive informationflow control based on program dependence graphs. International Journal of Information Security, 8(6):399–422, Dec. 2009. Supersedes ISSSE and ISoLA 2006.

[63] 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, Jan. 1998.

[64] C. A. R. Hoare. An axiomatic basis for computer programming. Comm. of the ACM, 12(10):576–580, 1969.

[65] M. Johns. On javascript malware and related threats - web page based attacks revisited. Journal in Computer Virology, Springer Paris, 4(3):161–178, August 2008.

[66] M. Johns. Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting. PhD thesis, University of Passau, 2009.

[67] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In SSP, 2011.

[68] B. Köpf and D. A. Basin. An information-theoretic model for adaptive side-channel attacks. In ACM Conference on Computer and Communications Security, pages 286–296, 2007.

[69] 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.

[70] 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).

[71] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973. Reprinted in J. of Computer Security, vol. 4, no. 2–3, pp. 239–263, 1996.

[72] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In Proc. Asian Computing Science Conference (ASIAN'06), volume 4435 of LNCS. Springer-Verlag, 2006. To appear.

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

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

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

[76] 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.

[77] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In Proceedings of IEEE Security and Privacy'10. IEEE, 2010.

[78] 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), Apr. 2010.

[79] J. Magazinius, A. Askarov, and A. Sabelfeld. Decentralized delimited release. In Proc. Asian Symp. on Programming Languages and Systems, LNCS. Springer-Verlag, Dec. 2011.

[80] J. Magazinius, A. Askarov, and A. Sabelfeld. Decentralized delimited release. Technical report, Chalmers University of Technology, 2011.

[81] 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.

[82] 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), Sept. 2010.

[83] 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, Mar. 2007.

[84] T. Mayfield, J. E. Roskos, S. R. Welke, J. M. Boone, and C. W. McDonald. Integrity in automated information systems. Technical Report P-2316, Institute for Defense Analyses, 1991.

[85] 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.

[86] J. McLean. A general theory of composition for a class of "possibilistic" security properties. IEEE Transactions on Software Engineering, 22(1):53–67, Jan. 1996.

[87] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized javascript. Whitepaper, http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf, January 2008.

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

[89] 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.

[90] 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.

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

[92] 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.

[93] D. Naumann. Theory for software verification. Draft, http://www.cs.stevens.edu/~naumann/pub/theoryverif.pdf, Jan. 2009.

[94] 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.

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

[96] P. Ørbæk. Can you trust your data? In Proc. TAPSOFT/FASE'95, volume 915 of LNCS, pages 575–590. Springer-Verlag, May 1995.

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

[98] C. P. Pfleeger and S. L. Pfleeger. Security in Computing (4th Edition). Prentice Hall, 2006.

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

[100] J. Rizzo and T. Duong. Padding oracles everywhere. http://ekoparty.org/juliano-rizzo-2010.php, 2010.

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

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

[103] 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, Sept. 2009.

[104] A. Sabelfeld and A. Birgisson. Multi-run security. In Proceedings of the European Symposium on Research in Computer Security (ESORICS), LNCS. Springer-Verlag, Sept. 2011.

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

[106] 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, Oct. 2004.

[107] 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.

[108] 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, Mar. 1999.

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

[110] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. J. Computer Security, Jan. 2009.

[111] R. S. Sandhu. On five definitions of data integrity, 1993.

[112] R. S. Sandhu. On five definitions of data integrity. In Proceedings of the IFIP WG11.3 Working Conference on Database Security VII, pages 257–267, 1994.

[113] F. B. Schneider. Enforceable security policies. ACM Transactions on Information and System Security, 3(1):30–50, 2000.

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

[115] K. Singh, A. Moshchuk, H. Wang, and W. Lee. On the incoherencies in web browser access control policies. In SSP, 2010.

[116] G. Smith. On the foundations of quantitative information flow. In Proc. Foundations of Software Science and Computation Structure, volume 5504 of LNCS, pages 288–302, Mar. 2009.

[117] 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, Jan. 1998.

[118] P. H. I. Systems. Sparkada examinar. Software release. http://www.praxis-his.com/sparkada/.

[119] M. Ter Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In 19th USENIX Security Symposium, 2010.

[120] R. van der Meyden. What, indeed, is intransitive noninterference? In Proc. European Symp. on Research in Computer Security, pages 235–250. Springer-Verlag, Sept. 2007.

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

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

[123] G. Winskel. The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge, MA, 1993.

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

[125] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In Proc. IEEE Computer Security Foundations Workshop, pages 29–43, June 2003.

[126] 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.

[127] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In Proc. IEEE Symp. on Security and Privacy, pages 236–250, May 2003.