Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D2.5

Report on safe information release

Due date of deliverable: 2008-08-31 (T0+36)

Actual submission date: 2008-10-15

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **CTH**

Revision (Unknown)

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to Chapter
CTH	1, 2, 5, 7
INRIA	3
IoC	6
TUD	4

This document was written by Aslan Askarov (CTH), Gilles Barthe (INRIA), Daniel Hedin (CTH), Peeter Laud (IoC), Alexander Lux (TUD), Heiko Mantel (TUD), Tamara Rezk (INRIA), Andrei Sabelfeld (CTH), and David Sands (CTH).

Executive Summary:
Report on safe information release

This document summarises deliverable D2.5 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr`.

Support for information-release, or *declassification*, policies is essential for practical use of information-flow security. The challenge of declassification is to distinguish intended information release from unintended leaks.

Following the goals of Task 2.2 to develop powerful policy frameworks for safe information release in expressive programming languages, the Mobius consortium has made significant contributions to the area of declassification. These contributions appear in diverse publication venues. This deliverable contains an overview of selected results and serves as a chart for Mobius' contributions to this area, including pointers to specific contributions for further details. The following results are in the main focus of the deliverable:

- To enhance understanding of declassification we have provided a road map to the area of information release. The classification of declassification policies according to *what*, *who*, *where* and *when* dimensions has helped clarify connections between existing models and facilitate the development of new ones. Our results are a step towards allowing combinations of policies from the individual dimensions into a solid *policy perimeter defence*. Perimeter defence is a standard security principle: as systems are no more secure than their weakest points, they must be defended in all dimensions of information release.

- Advancing the state of the art in policy perimeter defence, we have designed type-based enforcement mechanisms for the *what* and *where* dimensions of declassification for sequential and multi-threaded languages. We have constructed such mechanisms for both high- and low-level code.

- We have developed an approach to tracking information flow in the presence of cryptographic operations, and proposed a treatment of cryptographic operations where attackers may not learn useful information from ciphertexts. We have shown that this model naturally connects to computational adversary models. Further, we have devised a security type system that provably and straightforwardly enforces our security condition for a language that includes cryptographic primitives and message passing.

Contents

Chapter 1

Introduction

Support for information-release, or *declassification*, policies is essential for practical use of information-flow security. The need for declassification was emphasised at the requirements gathering stage of the Mobius project [47]:

> "The MIDP information-flow case strongly suggests the need for declassification policies."

Traditional *noninterference* [32] policies are an important building block for specifying end-to-end security (and they have served as such in Mobius' work on information flow [48]). However, noninterference flatly rejects programs with intentional information release. This means that some useful scenarios need policy support beyond noninterference. For example, releasing the average salary from a secret database of salaries might be needed for statistical purposes. Another example of deliberate information release is information purchase: specified secret information is revealed once a condition (such as "payment transferred") has been fulfilled. Yet another example is a password checking program: some information is released even if a log-in attempt fails, as the attacker learns that the attempted sequence is *not* the password. Releasing an encrypted secret is another ubiquitous example of intended information release.

A principal security concern for systems permitting information release is whether this release is safe; is it possible that an attacker compromises the mechanism for information release and extracts more secret information than intended? In the examples above, can individual salaries be (accidentally or maliciously) released to the attacker in the average salary computation? Can the attacker break an information purchase protocol to extract sensitive information before the payment is transferred? Is it possible that along with the result of password matching some other secret information is sneaked to the attacker? Might a program accidentally reveal plaintext instead of ciphertext to an attacker?

The Mobius consortium has made significant contributions to the area of declassification. These contributions appear in diverse publication venues. This deliverable contains an overview of selected results and serves as a chart for Mobius' contributions to this area, including pointers to specific contributions for further details.

Chapter 2 presents an overview of declassification, by classifying the basic goals according to the *dimensions of declassification* [61, 62]: *what* information is released, *who* releases information, *where* in the system information is released and *when* information can be released. Chapter 3 develops the *what* and *where* dimensions of declassification. It also suggests enforcement mechanisms that extend type systems [48, 49, 50] for noninterference for sequential languages to address declassification policies [13]. Chapter 4 considers the *what* and *where* dimensions of declassification for multi-threaded languages. It suggests type-based enforcement for a bytecode-like language with concurrency, which is built on top of an expressive declassification framework [43]. Chapter 5 addresses the challenge of reasoning about information flow in the presence of encryption. It develops *cryptographically-masked flows* by conservatively extending noninterference to allow safe encryption, decryption, and key generation, and provides type-based enforcement [4, 5]. Chapter 6 bridges cryptographically-masked flows with computational security. It proves that some natural properties of underlying cryptographic primitives are sufficient to guarantee computational security for programs that satisfy the security condition of cryptographically-masked flows [37]. Chapter 7 concludes.

Chapter 2

Dimensions of declassification [62]

The state of the art in information release comprises a fast growing number of definitions and analyses for different kinds of declassification policies over a variety of languages and calculi. Unfortunately, the relationship between different definitions of information release is often unclear and, in our opinion, the relationships that do exist between methods are often inaccurately portrayed. This creates hazardous situations where policies provide only partial assurance that information release mechanisms cannot be compromised.

For example, consider a policy for describing *what* information is released. This policy stipulates that at most four digits of a credit card number might be released when a purchase is made (as often needed for logging purposes). This policy specifies *what* can be released but says nothing about *who* controls which of the numbers are revealed. Leaving this information unspecified leads to an attack where the attacker launders the entire credit card number by asking to reveal different digits under different purchases.

To address this problem, we provide a road map of the main declassification definitions in current language-based security research (as a timely update on security policies from a survey on language-based information-flow security [57]). We classify the basic declassification goals according to four axes: *what* information is released, *who* releases information, *where* in the system information is released and *when* information can be released. Our classification includes attempts to outline connections between hitherto unrelated methods, as well as mark some clear distinctions, seeking to crystallise the security assurance provided by some known approaches. This chapter is an overview of the full version of the classification [61, 62] that includes detailed discussion of the dimensions and coverage of the state-of-the-art literature.

What *Partial*, or selective, information flow policies [22, 23, 33, 60, 29, 30, 43, 8, 13] regulate *what* information may be released. Partial release guarantees that only a part of a secret is released to a public domain. Partial release can be specified in terms of precisely which parts of the secret are released, or more abstractly as a pure *quantity*. This is useful, for example, when partial information about a credit card number or a social security number is used for logging.

A number of partial information flow policies can be uniformly expressed by using *equivalence relations* to model attacker knowledge (or, perhaps more precisely, to model attacker uncertainty). Suppose that the values of a particular secret range over integers (`int`), and that the value of the secret is not fixed—it is a parameter of the system. Without fixing a particular value for the secret, one way to describe how much an attacker knows (or can learn) about the secret is in terms of an equivalence relation. In this approach an attacker's knowledge about the secret is modelled in terms of the attacker's ability to distinguish elements of `int`. If the attacker knows nothing about the secret then this corresponds to saying that, from the attacker's viewpoint, any value in `int` looks the same as any other value. This is captured by the equivalence relation *All* satisfying $\forall m, n \in \texttt{int}.\ m\ \mathit{All}\ n$. I.e., all values (or variations) of the secret look the same to the attacker. Knowledge about the secret can be modelled by other, finer, equivalence relations. For example, if the parity of a secret is to be released (and nothing else about the secret), then this knowledge corresponds

to a partition of the domain into the even and the odd integers, i.e., the relation *Parity* satisfying:

$$m \; Parity \; n \iff m \; \texttt{mod} \; 2 = n \; \texttt{mod} \; 2$$

Thus an attacker cannot distinguish any two elements in the same equivalence class of *Parity*, because at most the parity is known. At the other extreme, total knowledge of the secret corresponds to the identity relation *Id*.

Suppose that we wish to express that a system leaks no more than the parity of a given secret, then we assume that the attacker already knows the parity, and show that nothing *more* is learned. This is expressed by saying that if we have any two possible values of the secret, m and n, such that $m \; Parity \; n$, then the attacker-observable results of running the system will be identical for these secrets. More precisely, if s is a function of type $\texttt{int} \to \texttt{int}$ that models, for particular public inputs, how the system maps the value of the secret to the observable output, then we write $s{:}Parity{\Rightarrow}Id$, meaning that

$$\forall m, n. m \; Parity \; n \Longrightarrow s(m) \; Id \; s(n)$$

In this notation standard noninterference (zero information flow) property corresponds to $s{:}All{\Rightarrow}Id$.

Several approaches to declassification can be understood (at least in part) in terms of the "equivalence class" approaches—even though at first glance they appear to be of a rather different nature. This includes the popular *delimited release* [57] and *relaxed noninterference* [38] policies.

Under the category "what" we also include properties which are abstractions of "what." One extreme abstraction is to consider the *quantity* of information released. Thus we consider "how much" to be an abstraction of "what." The most direct representation of this idea is perhaps the information-theoretic approach by Clark et al [20], which aims to express leakage in terms of an upper bound on the number of information-theoretic bits. The approach of Lowe [40] can be thought of as an approximation of this in which we assume the worst-case distribution. With this approximation the measure corresponds, roughly, to counting the number of equivalence classes in an equivalence-relation model. For a framework that integrates attacker belief into the analysis of quantitative information flow in a language-based setting see recent work by Clarkson et al. [21].

Who It is essential to specify *who* controls information release in a computing system. Ignoring the issue of control opens up attacks where the attacker "hijacks" release mechanisms to launder secret information. Myers and Liskov's *decentralised label model* [52] is based on ownership annotations for data. Several approaches use the decentralised label model as a starting point for semantic security conditions (e.g., [63, 53]. See, e.g., [28, 54, 12] for further ways of combining information flow and access control.

Where *Where* in a system information is released is an important aspect of information release. By delegating particular parts of the system to release information, one can ensure that no other (potentially untrusted) part can release further information. Considering *where* information is released, we identify two principal forms of locality: *code locality* policies (e.g., [56, 51, 17, 44, 26, 27, 2, 43, 13]) describing where physically in the code information may leak and *level locality* policies (e.g., [44, 43]) describing where information may flow relative to the security levels of the system.

When The fourth dimension of declassification is the temporal dimension, pertaining to *when* information is released. We identify three broad classes of temporal release specification: *Time-complexity based* (e.g., [65, 64, 34, 35, 39, 46]): information will not be released until certain time at the earliest. Time is an asymptotic notion typically relative to the size of the secret. *Probabilistic* (e.g., [9, 25]): with probabilistic considerations one can talk about the probability of a leak being very small per run, which implies that the adversary only can learn useful information when rerunning the program a certain number of times. *Relative* (e.g., [31, 19]): a non-quantitative temporal abstraction involves relating the time at which downgrading may occur to other actions in the system. For example: "downgrading of a software key may occur after confirmation of payment has been received."

Chapter 3

The *what* and *where* of declassification [13]

In this chapter we describe a modular method, proposed in [13], for building sound type systems for declassification from sound type systems for noninterference, and the instantiation of this method to a sequential fragment of the Java Virtual Machine.

We consider a declassification policy that combines the *what* and *where* dimensions, called *delimited non-disclosure* (DND), that is closely related to the *non-disclosure* policy proposed by Almeida Matos and Boudol [2], and to *localised delimited release* proposed by Askarov and Sabelfeld [8]. Informally, delimited non-disclosure is intended to combine two dimensions in a single construct of the form

$$\mathsf{declassify}\ e : \tau\ \mathsf{in}\ c$$

where e is an expression, τ is a security level (see below), and c is a statement.

The choice of the delimited non-disclosure policy is motivated by modularity; our aim is not to propose a new declassification policy but rather to propose a systematic extension of sound noninterference type systems to sound declassification type systems. This modularity is the most distinctive feature of our work. Existing proofs of soundness for declassification type systems proceed by induction over typing derivations, and reproduce a large part of the proof of soundness of the declassification-free fragment of the type system. Instead, we take advantage of the fact that delimited non-disclosure coincides with noninterference for programs without declassification; showing that if the type system enforces noninterference on the declassification-free fragment of the language, then it enforces delimited non-disclosure. In order to show the independence of our method w.r.t. the programming language to which it is applied, we describe the method based on a successor relation between program points, rather than on the syntactic structure of programs, and rely on the idea of control dependence regions, which over-approximate the scope of branching instructions (MOBIUS deliverable D2.3 [49]), i.e., of instructions that have two or more successors and that can thus yield implicit flows.

In expressing noninterference policies, it is common to model confidentiality clearances by a security lattice $(\mathcal{S}, \leq)$ whose elements represent the distinct confidentiality levels. Then, the expected security behaviour of a program is captured by a single global policy Γ that assigns confidentiality levels to variables. For expressing declassification policies, we take a simple generalisation of the noninterference setting: instead of a single global policy Γ for the program, we assume that there is a local policy $\Gamma[i]$ for each program point. (There are correctness conditions that should be satisfied by the family $\Gamma[i]$.) The use of local policies for the specification of declassification permits more precision on what is declassified within a code fragment, which substantially leverages the applicability of information flow type checking and supports more permissive policies. Local policies allow us to specify when the security level of a variable x is downgraded from its original policy.

A second distinctive feature of our approach is its application to low-level languages. In the context of mobile code security, it is essential for code consumers to be able to verify security policies independently

and efficiently on the code they receive. Since mobile code applications are typically downloaded in the form of bytecode programs, it is required that verification operates at this level. However, a large body of existing work on language-based security focuses on source languages; in fact we are not aware of any sound type system that supports declassification policies for unstructured languages.

Our method has been described and instantiated in a representative, but simplified, setting. Leveraging it to the sequential fragment of the Java Virtual Machine does not pose any major difficulty, but involves significant technicalities. Fortunately, these technicalities, linked to intrinsic object-oriented aspects, exceptions, and analyses precision, were already handled in the noninterference work on the JVM, MOBIUS deliverable D2.3 [49].

Information flow type systems are complex mechanisms whose soundness proofs are particularly involved, especially when considering permissive declassification policies for real programming languages such as the JVM. As such type systems are designed to complement existing type systems for safety and lie at the heart of the Trusted Computing Base, it is fundamental that the type system and its implementation be correct. In our earlier work [14], we used the proof assistant Coq to formally verify the soundness of an information flow type system that ensures noninterference for a sequential fragment of the Java Virtual Machine. In addition to providing strong guarantees about the correctness of the type system, the formalisation serves as a basis for a Proof Carrying Code architecture whose correctness is mechanically checked. A distinctive feature of our architecture is that the type system is executable inside constructive higher order logic and thus can be used use for verifying certificates within Coq, or for extraction to obtain an OCaml implementation of a lightweight information flow checker. As compared to Foundational Proof Carrying Code [3], which is deductive in nature, our approach exploits the interplay between deduction and computation to support efficient verification procedures and compact certificates.

As a benefit of the modularity of our approach, we believe that it is possible to achieve a proof of soundness for our DND information flow type system for the JVM at moderate cost, extending the formalisation reported in [14]. The Coq development in [14] is organised in two parts: a generic part, that derives the soundness of the noninterference type system from *unwinding lemmas*, and a specific part, that establishes the unwinding lemmas for a particular language, operational semantics, and type system. We are confident that extending the generic part of the formalisation to accommodate DND is direct, as is proving the soundness theorem for the DND type system in an abstract setting. Nevertheless, we anticipate a fair amount of bookkeeping in instantiation of the generic part for the specific JVM: even if there is no conceptual difficulty in programming a bytecode verifier in Coq that enforces DND, the specification of the noninterference type system for the JVM is rather large, and extending its definition (even in the modular fashion) to DND—and thus supporting a local declassification policy per program point instead of a global policy—will be demanding.

Chapter 4

The *what* and *where* of declassification for multi-threaded programs [43]

Multi-threading is an important aspect in mobile code scenarios, because it is used to avoid lock-ups caused by expensive computations or blocking operations. The intrinsic difficulties of multi-threading for information flow security without declassification have been investigated for high-level languages, for instance in [59, 66]. The two main approaches are *strong security* [59] and *observational determinism* [66]. In Task 2.1 of MOBIUS, the intrinsic difficulties have been already investigated for low-level languages [49]. Here, we focus on aspects of declassification. Guided by the dimensions of declassification presented in Chapter 2, we consider the dimensions *what* and *where*.

To characterize information flow security in a multi-threaded setting, our approach follows the established approach of *strong security*. A program is defined to be secure if it is bisimilar to itself according to a step-wise bisimulation relation. In the case of strong security, such a bisimulation relation requires that the public information after each execution step is independent from confidential information before the respective execution step. The novel security properties with declassification defined in [43] build on work that introduces intransitive non-interference [55] to language-based security [44]. The novel bisimulation relations permit some dependence, i.e., exceptional information flow or declassification, but restrict it in the dimensions *what* and *where* as described below. Concerning the dimension *where*, both principle forms of locality from Chapter 2 are supported. The intended *code locality* can be specified by marking statements as declassification statements with surrounding brackets. The intended *level locality* can be specified by a flow relation $\rightsquigarrow$ which may be intransitive and which is specified in addition to the standard partially-ordered flow relation $\leq$ (see Chapter 3). Concerning the dimension *what*, permissible declassification can be specified by *escape hatches* [58]. Escape hatches are pairs of a security level and an expression, as it might appear on the right-hand side of assignment statements in programs. The security property *WHERE* in [43] is defined using a bisimulation relation that restricts declassification to statements marked as declassification statements and permits information flow by declassification only along $\rightsquigarrow$. The security properties $WHAT_1$ and $WHAT_2$ in [43] are defined using a step-wise bisimulation relation that permits public information to depend on the current values of escape hatch expressions (but not on any other information derived from secrets). Further, the bisimulation requires that the values of escape hatch expressions during a computation may themselves only depend on public information and on the values of other escape hatch expressions. The difference between $WHAT_1$ and $WHAT_2$ is that the former requires this restriction for all escape hatches, while the later only requires this restriction for escape hatches whose expressions actually occur in a given program. The motivation for defining two different properties is elaborated in the following.

A main challenge for controlling declassification in the dimension *what* is to prevent *information laundering*, i.e., the change of a value supposed to be declassified depending on information that must not be declassified. For multi-threaded programs the treatment of information laundering is even more complicated than for sequential programs, since information may be written to and declassified from a variable in shared memory by parallel threads. For instance, consider the program consisting of the two threads `h2 := 0` and

$[\mathtt{l} := \mathtt{h1} + \mathtt{h2}]$, where the variable `l` is public and the variables `h1` and `h2` are confidential. Suppose we want to permit declassification of the value of $\mathtt{h1} + \mathtt{h2}$ (i.e., $(\mathit{public}, \mathtt{h1} + \mathtt{h2})$ is an escape hatch). If the thread $\mathtt{h2} := 0$ is executed first, the program declassifies the value of $\mathtt{h1} + 0$, i.e., the initial value of `h1`, but not the initial value of the sum, and hence should be rejected as insecure (assuming that $(\mathit{public}, \mathtt{h1})$ is not an escape hatch). By requiring that for each step values of escape hatch expressions may only depend on public information and the values of other escape hatch expressions, public output only depends on the initial values of escape hatch expressions and public input. Hence, both WHAT_1 as well as WHAT_2 prevent information laundering and correctly reject the program above.

In [43] we show that it is impossible to define a security property that provides adequate control in the dimension *what* with prevention of information laundering that is both compositional (w.r.t. sequential and parallel composition) as well as compliant with the principle *monotonicity of release*. Monotonicity of release [61, 47, 62] states that adding declassification annotations cannot make a given program insecure, if it is secure without these declassification annotations. The fundamental conflict between compositionality and monotonicity of release led us to defining two security properties for controlling *what* with different virtues: while the security property WHAT_1 is compositional, the security property WHAT_2 complies to monotonicity of release.

Our type system for the novel security properties is based on compositionality results of the bisimulation relations for syntactic program constructs, i.e., sequential composition, composition by control statements, and, particularly important for multi-threading, parallel composition. The type system contains rules for statements marked as declassification statements that permit declassification as specified by the intransitive flow relation $\rightsquigarrow$ and the escape hatches. We have proved the soundness of the type system with respect to the three novel security properties *WHERE*, WHAT_1, and WHAT_2.

Defining and enforcing the security properties from [43] not only for high-level but also for bytecode programs requires only few adaptations and design decisions, as presented in [41]. MOBIUS deliverable D2.3 [49] presents a type system for bytecode, including an extension to enforce the strong security property. Since *WHERE*, WHAT_1, and WHAT_2 are based on strong security, this type system serves as a basis for a sound type-based analysis of bytecode programs with respect to those security properties. The type system from [49] is modified by adding rules to permit declassification, inspired by the rules in [43]. Concerning *WHERE*, we decide to only allow the instructions `load` and `store` to be marked as declassification instructions. These two instructions directly move data from variables to the operand stack or vice versa. As for DND [13] in Chapter 3, security levels are specified for variables and elements on the operand stack. A marked instruction `load` can declassify a value from a variable to the top element of the operand stack, whereas a marked instruction `store` can declassify a value from the top element of the operand stack to a variable. In bytecode, there are further instructions that are candidates to be marked as declassification instructions. However, we opt for giving up some flexibility for a stricter marking discipline in the hope of making the marking less error-prone. For instance, `getfield` or `putfield`, which move data from fields of objects to the operand stack or vice versa, potentially do not only leak the moved value, but also the identity of the object whose field is accessed. Concerning WHAT_1 and WHAT_2, we define a language for specifying escape hatches, since bytecode does not have a suitable sub-language for escape hatch expressions. To prevent information laundering, the type system imposes restrictions on calculations on the operand stack.

In summary, we define a type system for the combined control of the dimensions *where* and *what* of declassification in multi-threaded programs. We prove the soundness of the type system according to novel security properties for each of the two dimensions. In addition, we formalize *prudent principles of declassification* [43], introduced informally as a sanity check for security properties in [61, 47, 62], and use them in order to argue about the adequacy of the novel security properties. We prove compliance of the security properties with the formalized principles, with the exception of monotonicity of release in the case of WHAT_1 (see above). We also provide compositionality results to facilitate a modular analysis, and to simplify the soundness proof for the type system. To complement the control of the dimensions *what* and *where* of declassification as characterized by the security properties *WHERE*, WHAT_1, and WHAT_2, we provide a security property to characterize the control of the dimension *who* in [42].

Chapter 5

Cryptographically-masked flows [5]

Cryptographic operations are ubiquitous in security-critical systems. Reasoning about information flow in such systems is challenging because typical information-flow definitions allow no flow from secret to public data. As discussed in Chapter 1, the latter requirement underlies *noninterference* [32], which demands that public outputs are unchanged as secret inputs are varied. While traditional noninterference breaks in the presence of cryptographic operations, the challenge is to distinguish between breaking noninterference because of legitimate use of sufficiently strong encryption and breaking noninterference by unintended leaks.

A common approach to handling cryptographic primitives in information-flow aware systems is by allowing declassification of encryption results. However, as discussed in Chapter 2, declassification is a versatile mechanism: different declassification dimensions correspond to different reasons why information is released. Attempts at framing cryptographically-masked flows into different dimensions have been made although not always with satisfactory results. For example, releasing the difference between two values of a secret whenever the results of its encryption are different can be a deceptive policy when assumptions about the underlying cryptographic primitives are not explicitly stated. If the underlying encryption function is bijective (assuming the key is fixed) then releasing the result of encryption is equivalent to releasing the secret itself. This phenomenon applies to typical policies from the *what* dimension, such as delimited release [57].

In essence, cryptographic primitives are small probabilistic algorithms that are supposed to satisfy certain complex security properties. Reasoning about programs using cryptographic primitives requires one to deal with those complex properties at some point. An attractive approach is to *modularise* the argument: to replace the cryptographic primitives with some operations, that operate in a similar way, but use simpler mechanisms (for example, no probabilistic steps) and have simpler security definitions. The actual implementability of those operations is not important. Still, the level of similarity between the abstract operations and the original cryptographic primitives must be such, that the statements derived for the modified program would convince us in similar statements holding for the original program.

This chapter introduces modelling-level non-probabilistic cryptographic primitives into an information-flow setting while preserving a form of noninterference property. The goal is to allow public access to ciphertexts obtained with a secret key, while modelling that the attacker may not learn anything useful from distinguishing such ciphertexts. This is achieved by building into the model a basic assumption that attackers may not distinguish between these ciphertexts and that decryption using the wrong key fails. This model has a close connection to probabilistic encryption and it naturally connects to computational adversary models. In Chapter 6 we prove that some natural properties of underlying cryptographic primitives (extended to hide key identities) are sufficient to guarantee computational security for programs that satisfy our possibilistic noninterference. This chapter is an overview of the full version [4, 5] that includes the formal details of the security condition, type system, a report from experience with a prototype and the proofs (formalised in the proof assistant Coq).

The idea behind our approach is illustrated in Figure 5.1, where dashed and solid lines correspond to secret and public values, respectively. Fixing some public (*low*) input z_L and varying secret (*high*) input from x_H to y_H may not reflect on a public output z'_L of a system that satisfies noninterference (illustrated in Figure 5.1(a)). Suppose the system in question involves encryption, such as in the program $z := \texttt{encrypt}\,(k, x)$ for some secret key k. Clearly, noninterference is broken: variation in the secret input

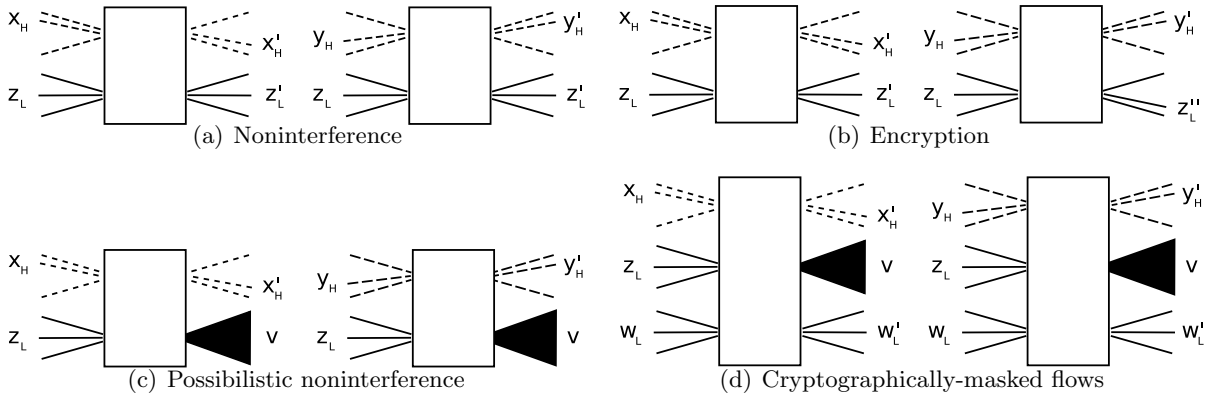

Figure 5.1: From noninterference to cryptographically-masked flows

from x_H to y_H may cause variation in the public output from z'_L to z''_L (illustrated in Figure 5.1(b)).

However, noninterference can be recovered if the result of encryption is *masked* by allowing it to be *any* value v. This means that variation of the high input from x_H to y_H does not affect the public output—any value v is a possible public output in both cases. This form of noninterference is known as *possibilistic noninterference* [45] (illustrated in Figure 5.1(c)). Overall, although low outputs might depend on low inputs and ciphertexts, no observation about possible low outputs may reveal information about changes in high inputs (illustrated in Figure 5.1(d)). The last figure illustrates that this kind of masking does not mask too much: low values that are not affected by encryption are distinguishable in the normal way.

We make a case for possibilistic noninterference as a natural model for cryptographically-masked flows. We show that a naive approach of collapsing all ciphertexts as indistinguishable opens up possibilities for *occlusion* [62], where masking an intended information flow in an indistinguishability definition may also mask other unintended leaks. The occlusion problem can be illustrated with the following example:

$$l_1 := \texttt{encrypt}(k, a);\ \texttt{if}\ h\ \texttt{then}\ l_2 := \texttt{encrypt}(k, b)\ \texttt{else}\ l_2 := l_1$$

where $\texttt{encrypt}(k, a)$ encrypts value a with secret key k. If all encrypted values are considered indistinguishable by attacker then we cannot distinguish between the two low variables l_1 and l_2 even though it is clear that the equality/inequality of the first and the second value reflects the secret value h. Therefore, we propose a finer indistinguishability relation that not only avoids occlusion but also, as shown in Chapter 6, guarantees computational security under some natural assumptions on the cryptographic primitives. With such a result at hand, our model allows focusing on enforcing a simple possibilistic property, which comes with a computational guarantee "for free."

Intuitively, we introduce cryptographic masking by assuming that the result of encryption is nondeterministic: it corresponds to a set of ciphertexts. Then, two commands are indistinguishable if for every pair of attacker-indistinguishable environments (containing the memories, key-generation, and I/O streams) in which the commands terminate it holds that there exists a *possibility* that each environment produced by the first command when run in the first environment can be produced by the second command when run in the second environment.

To demonstrate that enforcing possibilistic noninterference is straightforward, we have designed and implemented a security type system that provably enforces possibilistic noninterference for an imperative language with primitive cryptographic operations and communication channels. The type system prevents dangerous program behaviour (e.g., giving away a secret key or confusing keys or non-keys), which we exemplify with secure implementations of cryptographic protocols. Because the model is based on a standard noninterference property, it allows us to develop some natural extensions. In particular, we consider public-key cryptography and integrity, which accommodates reasoning about primitives that are vulnerable to chosen-ciphertext attacks. The main soundness result (that the type system indeed guarantees security) is based on our formalisation in the proof assistant Coq.

Chapter 6

Computational soundness of cryptographically-masked flows [37]

As discussed in the previous chapter, cryptographically-masked flows are an abstraction for the cryptographic primitive of symmetric encryption. Similarly to other abstractions of cryptographic primitives, for example the Dolev-Yao model (which is geared towards cryptographic protocols, where one typically considers only computational steps from a restricted set), one may ask whether the abstraction is *sound* — can we state a theorem saying that a program proven secure in the abstract setting is also secure (possibly in a somewhat different sense) in the real world? For the Dolev-Yao method, there exist results stating under which conditions the abstraction is sound. The conditions might, e.g., restrict the operations that may be performed with cryptographic data. The soundness result may also apply to a certain kind of security properties. For example, the preservation of integrity properties even under active attacks can be derived quite naturally for the Dolev-Yao model [24], as those properties are stated in the same way in the model and in the real world. Confidentiality properties are harder to carry over — in the Dolev-Yao model a value is either leaked fully or not leaked at all, while in the real world partial leaks are possible, too. In the results about confidentiality properties, one typically has to restrict the operations performed with the secret values, such that the absence of full leaks would also mean the absence of partial leaks [24, 10].

Cryptographically-masked flows are mainly used to argue about confidentiality properties. The definition of confidentiality is somewhat similar to the one in the real world. Both definitions consider the possible outputs for each input and require that the set / distribution of outputs does not depend on the secret inputs. Hence we may hope that modelling (computationally) secure information flow with cryptographically-masked flows is sound, and we have indeed obtained a result [37] to that effect. It states that if the program containing symmetric encryption has secure information flow in the sense of the previous chapter, the used encryption primitive is secure, and if the program uses the cryptographic data (keys and ciphertexts) in a "reasonable" manner, then the program also has computationally secure information flow in the real world. Also, one does not have to refer to the concrete semantics of the program (where encryption is implemented by a probabilistic operation) to verify whether the conditions on the usage of cryptographic data are satisfied, hence the computational security of the information flow of a program can be derived by the arguments referring only to the abstract semantics of the program (where encryption is modeled as a possibilistic operation, as described in the previous chapter).

The security properties we demand from the encryption primitive are the same as the ones conjectured as necessary in the paper introducing cryptographically-masked flows [4] — the encryption must be secure against chosen-plaintext attacks, hide the identities of keys and the lengths of plaintexts (this is called *type-0 security* by Abadi and Rogaway [1]) even in the presence of *key-dependent messages* [16]. The encryption primitive must also preserve the *integrity of plaintexts* [15], also in the presence of key-dependent messages. The latter property, which is absent from the earlier works on program analysis for computationally secure information flow [35] is necessary for handling the decryption.

The conditions we put on the program structure are the following. We allow only good keys to be used

as keys (which is obvious) and the keys themselves can only be used as encryption/decryption keys or in polymorphic operations. The polymorphic operations are those that only work with the structure of values, for example pairing and projections. Additionally we consider the plaintext argument of encryption and the ciphertext argument (as long as it is a ciphertext) of decryption to be polymorphic. This condition on keys will most probably not be an issue in the real world — there should be no need to perform computations on the actual bit-string that is the value of that key (although it is allowed in [35]). We also allow the ciphertexts to be used only in polymorphic operations. This may be a more serious issue — for example, we can say nothing about a program that, after computing a ciphertext and before sending it out, applies some transformation to it, for example an error-correcting code. Obviously, the conditions that we have stated for the usage of cryptographic data are sufficient, and maybe not necessary, but the examples we provide suggest that at least the conditions on the usage of ciphertexts will be hard to remove. Indeed, consider the program $\texttt{if lsb}(\texttt{encrypt}(k,a)) = 1 \texttt{ then } l := h \texttt{ else } l := \texttt{lsb}(\texttt{encrypt}(k,a))$, where h is a one-bit secret. Here $\texttt{lsb}(\texttt{encrypt}(\ldots))$ is basically a random-number generator outputting a single bit. In the abstract semantics, the possible final values of l are 0 and 1, no matter what the value of h is. Hence we consider that program secure. In the concrete semantics, l can still be both 0 and 1, but their probabilities depend on the value of h.

The aforementioned conditions essentially turn the model of cryptographically-masked flows into the Dolev-Yao model. Indeed, we have proposed yet another abstract semantics, closely resembling the Dolev-Yao model, that is also computationally sound under the same conditions. The new model extends the set of values (that are simply bit-strings even in the abstract semantics of cryptographically-masked flows) by $v ::= b \mid \mathsf{k}\langle i\rangle \mid \{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle} \mid (v_1, v_2)$, where b is a bit-string. The value $\mathsf{k}\langle i\rangle$ denotes the (formal) key produced by the i-th invocation of the key generation. Similarly, $\mathsf{r}\langle j\rangle$ is the formal randomness of the j-th ciphertext.

The semantics of a normal operation is a *deterministic* function from tuples of bit-strings to bit-strings; a normal operation can only be applied to bit-strings. The operations of pairing, projections, key generation, encryption and decryption make use of the structure of the values. Similarly to normal operations, they may be applied only to values of correct structure. Thanks to the conditions we have described before, operations with forbidden arguments are never attempted. Note that key generation and encryption are now deterministic operations, too (although dependent on the execution context that just counts the number of generated keys and ciphertexts).

The condition for secure information flow now simply states that a program must map two initial states differing only in their secret inputs to final states that are low-equivalent. Probably the most interesting part of the new model is the definition of low-equivalence of states. Two states are considered *low-equivalent* if their public parts are *equivalent* in the sense of Abadi and Rogaway [1]. To elaborate, given a state S, we form the tuple $(S(x_1), \ldots, S(x_t))$, where x_i ranges over all public variables in some fixed order. We then find the *pattern* of this tuple precisely as in [1]. In constructing the pattern of a formal message, if the Dolev-Yao attacker is incapable of deriving the key $\mathsf{k}\langle i\rangle$ from that message, then we replace all submessages of that message having the form $\{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle}$ with the message $\square^{\mathsf{r}\langle j\rangle}$. The message $\square^{\mathsf{r}\langle j\rangle}$ denotes an *undecryptable ciphertext*, made with the formal coins $\mathsf{r}\langle j\rangle$. Two states S and S' are low-equivalent if their patterns P and P' are equal modulo an α-conversion of formal keys and coins. Such an α-conversion consists of two permutations of integers, used to rename the indices of formal keys and coins, respectively.

The existing information flow analyses [35, 36] working directly on the concrete semantics can handle more operations (computing with keys and ciphertexts) than Dolev-Yao models. Cryptographically-masked flows, which do not abstract the structure of values, might also be expected to be more powerful than Dolev-Yao models. Unfortunately, the conditions described here will not let this difference (if it exists) to materialise.

Chapter 7

Conclusions

This chapter summarises the results and briefly discusses their impact and publications that have emerged from work on Task 2.2 on safe information release.

Results There is rich evidence that we have achieved the task goals of developing powerful policy frameworks for expressive programming languages:

- Seeking to enhance understanding of declassification, we have provided a road map to the area of information release. The classification of declassification policies according to *what*, *who*, *where* and *when* dimensions has helped clarify connections between existing models and facilitate the development of new ones. Our results are a step toward developing policies that allow combinations of policies from the individual dimensions into solid *policy perimeter defence*. Perimeter defence is a standard security principle: as systems are no more secure than their weakest points, they must be defended across the entire perimeter of the network. The ambition with policy perimeter is to prevent attackers from compromising the dimensions of information release with the weakest defence.
- Advancing the state of the art in building the policy perimeter defence, we have explored the *what* and *where* dimensions of declassification. As intended in the task goal, we have designed type-based enforcement mechanisms for the *what* and *where* of declassification for sequential and multi-threaded languages. We have constructed such mechanisms for both high- and low-level code.
- We have developed an approach to tracking information flow in the presence of cryptographic operations, based on possibilistic noninterference. We have argued that a possibilistic treatment of cryptographic operations leads to a natural model of attackers that may not learn useful information from ciphertexts. We have showed that this model naturally connects to computational adversary models. Further, we have devised a security type system that provably and straightforwardly enforces possibilistic noninterference for a language that includes cryptographic primitives and message passing.

Impact The results of the task have been published in prestigious venues [6, 18, 53, 4, 43, 8, 37, 13, 5, 62, 42] and have enjoyed impact both inside and outside of the project. Our declassification road map often acts as a starting point of research in declassification and a driving force for combining the dimensions of declassification. Several approaches to relating and combining dimensions of declassification have emerged within the project and beyond it. Controlling and combining the *what* and *where* dimensions has become a particularly lively area of research [43, 8, 11, 13]. We have explored the *when* dimension in our work on *flow locks*[18] and the *who* dimension in our work on *qualified robustness* [53]. Unifying declassification, encryption and key release policies is another area that has emerged from the project [7, 37, 5]. Treating information-release policies in low-level languages is another example of a previously unexplored area that has been pioneered by the project [13].

Further development within the project The results of the task serve as input for Tasks 2.6 on type system prototypes and 3.5 on combining type and logic-based techniques for security.

Bibliography

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. of Cryptology*, 15(2):103–127, 2002.

[2] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240, June 2005.

[3] A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.

[4] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. In *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, Seoul, Korea, August 2006. Springer-Verlag.

[5] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theoretical Computer Science*, 402:82–101, August 2008. Special issue for TGC 2006.

[6] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *European Symposium On Research In Computer Security*, number 3679 in Lecture Notes in Computer Science. Springer-Verlag, September 2005.

[7] 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.

[8] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, San Diego, California, June 2007.

[9] M. Backes and B. Pfitzmann. Intransitive non-interference for cryptographic purposes. In *Proc. IEEE Symp. on Security and Privacy*, pages 140–153, May 2003.

[10] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. In *Proc. IEEE Symp. on Security and Privacy*, pages 171–182, May 2005.

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

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

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

[14] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. European Symp. on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[15] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt 2000*, volume 1976 of *LNCS*, pages 531–545, January 2000.

[16] J. Black, P. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 62–75. Springer-Verlag, August 2002.

[17] A. Bossi, C. Piazza, and S. Rossi. Modelling downgrading in information flow security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 187–201, June 2004.

[18] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In P. Sestoft, editor, *European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer-Verlag, 2006.

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

[20] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

[21] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 31–45, June 2005.

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

[23] 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.

[24] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *Proc. ESOP'05*, volume 3444 of *LNCS*, pages 157–171. Springer-Verlag, April 2005.

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

[26] R. Echahed and F. Prost. Handling declared information leakage. In *Proc. Workshop on Issues in the Theory of Security*, January 2005.

[27] R. Echahed and F. Prost. Security policy in a declarative style. In *ACM International Conference on Principles and Practice of Declarative Programming*, pages 153–163, July 2005.

[28] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 130–140, May 1997.

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

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

[31] P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.

[32] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–22. IEEE Press, 1982.

[33] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[34] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 77–91. Springer-Verlag, April 2001.

[35] P. Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 159–173. Springer-Verlag, April 2003.

[36] P. Laud and V. Vene. A type system for computationally secure information flow. In *Proc. Fundamentals of Computation Theory*, volume 3623 of *LNCS*, pages 365–377, August 2005.

[37] Peeter Laud. On the computational soundness of cryptographically masked flows. In *POPL*, pages 337–348. ACM Press, 2008.

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

[39] P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 112–121, November 1998.

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

[41] A. Lux and H. Mantel. Controlling the what and where of declassification in bytecode. Submitted, August 2008.

[42] A. Lux and H. Mantel. Who can declassify? In *Preproceedings of FAST*, 2008.

[43] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007*, number 4421 in Lecture Notes in Computer Science, pages 141–156. Springer-Verlag, 2007.

[44] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Asian Programming Languages and Systems Symposium*, LNCS 3303, pages 129–145, Taipei, Taiwan, November 2004. Springer-Verlag.

[45] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symp. on Security and Privacy*, pages 177–186, May 1988.

[46] J. C. Mitchell. Probabilistic polynomial-time process calculus and security protocol analysis. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 23–29. Springer-Verlag, April 2001.

[47] MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements, 2006. Available online from `http://mobius.inria.fr`.

[48] MOBIUS Consortium. Deliverable 2.1: Intermediate report on type systems, 2006. Available online from `http://mobius.inria.fr`.

[49] MOBIUS Consortium. Deliverable 2.3: Report on type systems, 2007. Available online from `http://mobius.inria.fr`.

[50] MOBIUS Consortium. Deliverable 3.2: Intermediate report on embedding type-based analyses into program logics, 2007. Available online from `http://mobius.inria.fr`.

[51] J. Mullins. Non-deterministic admissible interference. *J. of Universal Computer Science*, 6(11):1054–1070, 2000.

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

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

[54] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. ACM International Conference on Functional Programming*, pages 46–57, September 2000.

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

[56] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 214–227, June 1999.

[57] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, 2003.

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

[59] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop*, pages 200–215. IEEE Press, 2000.

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

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

[62] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007.

[63] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 279–294. Springer-Verlag, April 2005.

[64] D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.

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

[66] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, USA, June 2003. IEEE Press.

Appendix A

Copies of Publications

Declassification: Dimensions and Principles*

Andrei Sabelfeld David Sands
Department of Computer Science and Engineering
Chalmers University of Technology and the University of Göteborg
412 96 Göteborg, Sweden
www.cs.chalmers.se/~{andrei, dave}

Abstract

Computing systems often deliberately release (or declassify) sensitive information. A principal security concern for systems permitting information release is whether this release is safe: is it possible that the attacker compromises the information release mechanism and extracts more secret information than intended? While the security community has recognised the importance of the problem, the state-of-the-art in information release is, unfortunately, a number of approaches with somewhat unconnected semantic goals. We provide a road map of the main directions of current research, by classifying the basic goals according to *what* information is released, *who* releases information, *where* in the system information is released and *when* information can be released. With a general declassification framework as a long-term goal, we identify some prudent *principles* of declassification. These principles shed light on existing definitions and may also serve as useful "sanity checks" for emerging models.

1 Introduction

Computing systems often deliberately release (i.e., *declassify* or *downgrade*) sensitive information. Without a possibility to leak secrets, some systems would be of no practical use. For example, releasing the average salary from a secret database of salaries is sometimes needed for statistical purposes. Another example of deliberate information release is information purchase. An information purchase protocol reveals the secret information once a condition (such as "payment transferred") has been fulfilled. Yet another example is a password checking program that leaks some information about the password. Some information is released even if a log-in attempt fails: the attacker learns that the attempted sequence is *not* the same as the password.

Information release is a necessity in these scenarios. However, a principal security concern for systems permitting information release is whether this release is safe. In other words, is it possible that the attacker compromises the mechanism for information release and extracts more secret information than intended? Applying this question to the examples above: can individual salaries be (accidentally or maliciously) released to the attacker in the average salary computation? Can the attacker break an information

*In *Journal of Computer Security*, © IOS Press.

purchase protocol to extract sensitive information before the payment is transferred? Is it possible that along with the result of password matching some other secret information is sneaked to the attacker? This leads to the following general problem:

> What are the policies for expressing intentional[1] information release by programs?

Answering this question is a crucial challenge [67, 78] for information security. Because many systems rely on information release, we believe that answering this question satisfactorily is the key to enabling technology transfer from existing information security research into standard security practice.

While the security research community has recognised the importance of the problem, the state-of-the-art in information release comprises a fast growing number of definitions and analyses for different kinds of information release policies over a variety of languages and calculi. The relationship between different definitions of release is often unclear and, in our opinion, the relationships that do exist between methods are often inaccurately portrayed. This creates hazardous situations where policies provide only partial assurance that information release mechanisms cannot be compromised.

For example, consider a policy for describing *what* information is released. This policy stipulates that at most four digits of a credit card number might be released when a purchase is made (as often needed for logging purposes). This policy specifies *what* can be released but says nothing about *who* controls which of the numbers are revealed. Leaving this information unspecified leads to an attack where the attacker launders the entire credit card number by asking to reveal different digits under different purchases.

This article does *not* propose any new declassification mechanisms. Instead we focus on the variety of *definitions* of security, in a language-based setting, which employ some form of declassification. We do not study specific proof methods, program logics, types systems or other static analysis methods. The contributions of this article are twofold:

- Firstly, we provide a road map of the main declassification definitions in current language-based security research (as a timely update on security policies from a survey on language-based information-flow security [67]). We classify the basic declassification goals according to four axes: *what* information is released, *who* releases information, *where* in the system information is released and *when* information can be released. Our classification includes attempts to outline connections between hitherto unrelated methods, as well as mark some clear distinctions, seeking to crystallise the security assurance provided by some known approaches.

- Secondly, we identify some common *semantic principles* for declassification mechanisms:

  - *semantic consistency*, which states that security definitions should be invariant under equivalence-preserving transformations;

[1] Note that in this article we will refer to both *intentional*, meaning deliberate, and *intensional*, meaning the opposite of extensional, when discussing declassification.

- *conservativity*, which states that the definition of security should be a weakening of noninterference;
- *monotonicity of release*, which states that adding declassification annotations cannot make a secure program become insecure. Roughly speaking: the more you choose to declassify, the weaker the security guarantee; and
- *non-occlusion*, which states that the presence of declassifications cannot mask other covert information leaks.

These principles help shed light on existing approaches and should also serve as useful "sanity checks" for emerging models.

This article is a revised and extended version of a paper published in the IEEE Computer Security Foundations Workshop 2005 [71]. Compared to the earlier version, we overview some new work on declassification that has appeared under 2005 [35, 34, 50, 32, 37, 43], consider "why" and "how" as other possible dimensions of declassification, sketch challenges for enforcing declassification policies along the dimensions, discuss dimensions of endorsement (the dual of declassification for integrity), and make other changes and improvements throughout.

2 Dimensions of declassification

This section provides a classification of the basic declassification goals according to four axes: *what* information is released, *who* releases information, *where* in the system information is released and *when* information can be released.

2.1 What

Partial, or selective, information flow policies [15, 16, 38, 70, 27, 28] regulate *what* information may be released. Partial release guarantees that only a part of a secret is released to a public domain. Partial release can be specified in terms of precisely which parts of the secret are released, or more abstractly as a pure *quantity*. This is useful, for example, when partial information about a credit card number or a social security number is used for logging.

The PER model of information A number of partial information flow policies can be uniformly expressed by using equivalence relations to model attacker knowledge (or, perhaps more precisely, to model attacker uncertainty). Here we outline this idea (and where it has been used previously), before we go on to show how some recent approaches to declassification can be understood in these terms.

Suppose that the values of a particular secret range over `int`, and that the value of the secret is not fixed—it is a parameter of the system. Without fixing a particular value for the secret, one way to describe how much an attacker knows (or can learn) about the secret is in terms of an equivalence relation. In this approach an attacker's knowledge about the secret is modelled in terms of the attacker's ability to distinguish elements of `int`. If the attacker knows nothing about the secret then this corresponds to saying that,

from the attacker's viewpoint, any value in `int` looks the same as any other value. This is captured by the equivalence relation All satisfying $\forall m, n \in \texttt{int}.m\ All\ n$. I.e., all values (or variations) of the secret look the same to the attacker. Knowledge about the secret can be modelled by other, finer, equivalence relations. For example, if the parity of a secret is to be released (and nothing else about the secret), then this knowledge corresponds to a partition of the domain into the even and the odd integers, i.e., the relation $Parity$ satisfying:

$$m\ Parity\ n \iff m \bmod 2 = n \bmod 2$$

Thus an attacker cannot distinguish any two elements in the same equivalence class of $Parity$, because at most the parity is known. At the other extreme, total knowledge of the secret corresponds to the identity relation Id.

This model of information extends to a model of information flow by describing how systems transform equivalence relations. As shown in [70], this is equivalent to Cohen's *selective dependency* [15, 16], and is related to the so-called *unwinding conditions* known from Goguen and Messeguer's work on noninterference and its descendants [30, 31].

It is worth remarking that noninterference in this paper is mostly concerned with protecting the secret (*high*) part of memory from the attacker who can observe the public (*low*) part of memory. This view follows the data protection view of noninterference, as it is often used in language-based security [16, 77, 67]. This is somewhat different from the interpretation of noninterference in event-based systems that is concerned with protecting the occurrence of secret events from public-level observers [25, 46, 65]. For the relation between these views, see [48, 26].

Suppose that we wish to express that a system leaks no more than the parity of a given secret, then we assume that the attacker already knows the parity, and show that nothing *more* is learned. This is expressed by saying that if we have any two possible values of the secret, m and n, such that $m\ \ Parity\ \ n$, then the attacker-observable results of running the system will be identical for these secrets. More precisely, if $s : \texttt{int} \to \texttt{int}$ models, for particular public inputs, how the system maps the value of the secret to the observable output, then we write $s{:}Parity{\Rightarrow}Id$, meaning that

$$\forall m, n.m\ Parity\ n \Longrightarrow s(m)\ Id\ s(n)$$

In this notation standard noninterference (zero information flow) property corresponds to $s{:}All{\Rightarrow}Id$.

The use of explicit equivalence relations to model such dependencies appears in several places. In the security context it was first introduced by Cohen. Also in the information flow context, the mathematical properties of the lattice of equivalence relations was explored by Landauer and Redmond [39]. Partial equivalence relations generalise the picture by dropping the reflexivity requirement. A partial equivalence relation (PER) over some domain D is just an equivalence relation on E such that $E \subseteq D$. Hunt [36], inspired by work in the semantics of typed lambda calculi, introduced the use of the lattice of PERs as a general static analysis tool, and showed that they could be incorporated into a classic abstract interpretation framework. Abadi et al. [3] (as well as Prost [62]) use partial equivalence relations in essentially the same

way to argue the correctness of dependency analyses. The present authors [70] showed how the PER model can also be extended to reason about nondeterministic and probabilistic systems, and also showed that not only does the PER model generalise Cohen's framework but also other formulations such as Joshi and Leino's logical formulation, including the use of *abstract variables* [38].

Recently, *abstract noninterference* [27, 28] has been introduced by Giacobazzi and Mastroeni. The properties expressible using *narrow abstract noninterference* are similar to those expressible using partial equivalence relation models. The relationship between the two approaches is not completely straightforward, however. On the one hand abstract noninterference is based on the general concept of a closure operator, so can also represent classic abstract interpretations. [2]

On the other hand the use of partiality in the PER setting—useful for example in describing security properties of higher-order functions, as well as security properties of the system as a whole—cannot be directly represented in the abstract noninterference setting.

Clark et al. [12] suggest more concretely how abstract noninterference relates to the PER model, and notably how nondeterminism is needed to model the notion of *weak observers*—observers who cannot see all the low values in the system. The relation between the two approaches is made more concrete in recent work by Hunt and Mastroeni [37], where it is shown that information flow properties expressible with a particular class of (total) equivalence relations can be captured by narrow abstract noninterference (NNI). It is also shown that, at least in a technical sense, NNI is equivalent to the equivalence relation model but the more general form of abstract noninterference is strictly more expressive than the equivalence relation properties. The difference lies in the attacker model. Implicitly in the deterministic PER model an attacker is assumed to gain information by observing individual runs of the system, whereas in the NNI model an attacker can be modelled as observing abstractions of sets of runs: for example, an attacker who can only observe interval approximations of any given set of integer results.

The abstract noninterference framework allows for the derivation of the most powerful attacker model for which a given program is secure. This is achieved by either hiding public output data (as little as possible) [27] or by revealing secret input data (also as little as possible) [28]. Both cases contribute to the attacker's model of data indistinguishability: the former is concerned with the indistinguishability of the output while the latter treats the indistinguishability of the input. Yet Giacobazzi and Mastroeni refer to the former as "attacker models" (which is asserted to fall into the "who" class [50]) and to the latter—somewhat surprisingly—as "declassification" (which is asserted to fall into the "what" class [50]). The latter is claimed to be adopted from robust declassification [79] by removing active attackers [27, 28]. This classification does not agree with ours because, in our view, robust declassification [79] addresses

[2] In the conference version of this paper we claimed that abstract noninterference could accurately represent disjunctions of properties such as "at most *one* of secrets A and B are leaked". We now believe that this is not the case. Note that there is a potential confusion of two notions of "property". Abstract noninterference assumes that an attacker observes the system via abstraction functions. One can, as standard in abstract interpretation, accurately represent disjunctive combinations of arbitrary abstract domains—i.e., disjunctive properties. But it does not follow from this that disjunctive *information flow* properties can be represented.

the question whether the declassification mechanism is robust against *active* attackers, and therefore is a "who" property (as opposed to "what"). The key property of robust declassification is that active attackers may not manipulate the system to learn more about secrets than passive attackers already know. When there are no active attackers (as in partial release), declassification is vacuously robust. We refer to the more recent paper by Mastroeni [50] which casts more light on this matter.

The basic idea of partial release based on (partial) equivalence relations is simple and attractive. As we shall see later, the fact that it is essentially an *extensional* definition means that it is semantically well behaved.

Related approaches In the remainder of this section we argue that two other recent approaches to declassification can be understood (at least in part) in terms of the "equivalence class" approaches—even though at first glance they appear to be of a rather different nature.

Delimited release Recent work by Sabelfeld and Myers [68] introduces a notion called *delimited release*. It enables the declassification policy to be expressed in terms of a collection of *escape hatch* expressions which are marked within the program via a `declassify` annotation. This policy stipulates that information may only be released through escape hatches and no additional information is leaked.

More precisely stated, a program satisfies delimited release if it has the following property: for any initial memory state s and any state t obtained by varying a secret part of s, if the value of all escape-hatch expressions is the same in both s and t, then the publicly observable effect of running the program in state s and t will be the same.

Interestingly, this definition does not demand that the information is *actually* released via the declassify expressions (even though the specific type system does indeed enforce this)—only that the declassify expressions within the program form the policy. As a rather extreme example, consider

$$\texttt{if true then } l := h * h \texttt{ else } l := \texttt{declassify}(h * h)$$

This satisfies delimited release: the secret characterised by the expression $h * h$ is released to the public variable l, i.e., if we have two memories which differ only in h and for which the respective values of $h * h$ are equal, then the respective final values of l after running the above program will also be equal. This example illustrates that even when we know *what* information is released, it may be useful to also know *where* it is released.

To see how delimited release relates to the PER model, we note that every expression (or collection of expressions) over variables in some memory state induces an equivalence relation on the state. For any expression e, let $[\![e]\!]$ denote the corresponding (partial) function from states to some domain of values. Let $\langle e \rangle$ denote the equivalence relation on states $(s, t, \ldots)$ induced by $[\![e]\!]$ as follows:

$$s \langle e \rangle t \iff [\![e]\!]s = [\![e]\!]t$$

We generalise this to a set of expressions E as

$$s \langle E \rangle t \iff \forall e \in E. [\![e]\!]s = [\![e]\!]t$$

If we restrict ourselves to the two-point security lattice (for simplicity) the delimited release property can be expressed in the PER model as:

> If E is the set of declassify expressions in the program, then for all memory states s and t such that the low parts of s and t are equal, and such that $s\langle E\rangle t$, then the respective low observable parts of the output of running the program on s and t are equal.

Sabelfeld and Myers also point out that delimited release is more general than Cohen's simple equivalence relation view: declassification expressions may combine both high and low parts of the state. This provides a form of *conditional release*.[3] For example, to express the policy: "declassify h only when the initial value of l is non-zero" we can just use the expression $\texttt{declassify}(h * l)$ since from this the low observer can always reconstruct the value of h—except when l is zero. A more general form of conditional release is specified by expressions such as

$$\texttt{if}\ (\mathit{payment} > \mathit{threshold})\ \texttt{then}\ \mathit{topsecret}\ \texttt{else}\ \mathit{secret}$$

with the guarantee that the sensitive information stored in $\mathit{topsecret}$ is released if the value of the public variable $\mathit{payment}$ is greater than some constant $\mathit{threshold}$. Otherwise, the less sensitive information secret is released.

Reflecting this idea back into the equivalence relation view, this just corresponds to the class of equivalence relations which are expressed in terms of the whole state and not just the high part of the state.

Syntactic escape hatches for characterising what can be leaked is a convenient feature of delimited release. It is however worth highlighting that the purpose of an escape hatch is merely to define indistinguishability. For example, policies defined by escape hatch expressions e and $f(e)$ (for some bijective function f) are equivalent. Indeed, they define the same indistinguishability relations because $[\![e]\!]s = [\![e]\!]t$ if and only if $[\![f(e)]\!]s = [\![f(e)]\!]t$. Interestingly, this implies, for example, that program $l := \texttt{declassify}(f(h)); l := h$ is secure for all bijective f.

It appears natural to declare $\texttt{encrypt}_k(\mathit{secret})$ and $\texttt{hash}(\mathit{pwd})$ as escape hatch expressions (cf. [68]). The intuition with such a declaration is that the result of encryption or hashing can be freely released. Clearly, if encryption and hashing functions are bijective then a laundering attack along the lines above is possible. If collisions are possible, exploiting this artifact becomes more involved. Nevertheless, suppose $\texttt{noncolliding}(x)$ is true whenever $\forall y.\texttt{hash}(x) = \texttt{hash}(y) \implies x = y$. Leaky program

$$l := \texttt{declassify}(\texttt{hash}(\mathit{pwd}));\ \texttt{if}\ \texttt{noncolliding}(\mathit{pwd})\ \texttt{then}\ l := \mathit{pwd}$$

is accepted by the delimited release definition. Clearly, these examples are possible because problematic invertability of encryption and hashing functions are outside the delimited release model.

[3] The conditional noninterference notion from [31] is a predecessor to the notion of intransitive noninterference discussed in Section 2.3 on "where" definitions.

Relaxed noninterference Li and Zdancewic [42] express downgrading policies by labelling subprograms with sets of lambda-terms which specify how an integer can be leaked.[4] They show that these labels form a lattice based on the amount of information that they leak. This is claimed to be closely related to *intransitive noninterference* (discussed below). Here, however, we argue that the lattice of labels from [42] is closely related to the lattice of equivalence relations, and thus the class of declassification properties that can be expressed is similar to the equivalence-relation class. The semantic interpretation of a label l [42][Def. 4.2.1] is closure operation:

$$\{g' \mid g' \equiv g \circ f,\ f \in l\}$$

Intuitively we can think of the meaning of a given $f \in l$ as an abstraction of all possible ways in which a program might use the result of the "leaky component" f.

In [42] the equality relation ($\equiv$) in the above definition is taken to be a particular decidable *syntactic* equivalence. We will refer to this original definition as an *intensional* interpretation of labels. To relate labels to the PER model we consider an *extensional interpretation* of labels in which $\equiv$ is taken to be semantic (extensional) equality.

We can map labels to equivalence relations (over the domain of secrets) using the same mapping as for delimited release: if l is a set of lambda-terms, each with an integer-typed argument, then we define the equivalence relation on integers as:

$$m\langle l\rangle n \iff \forall f \in l. fm = fn$$

Without going into a detailed argument, we claim that the extensional semantic interpretation of labels yields a sublattice of the lattice of equivalence relations. Note in particular (as with the intensional definition) that the top and bottom points in the lattice of labels are $H \equiv \{\lambda x : \texttt{int}.c\}$ (for some constant c) and $L \equiv \{\lambda x : \texttt{int}.x\}$, which following the construction above can easily be seen to yield the largest equivalence relation (All, which relates everything to everything) and the smallest (the identity relation Id), respectively.

Downgrading is specified by *actions* of the form $l_1 \overset{a}{\rightsquigarrow} l_2$. In the equivalence relation view this corresponds to saying, roughly, that the action a maps arguments related by $\langle l_1\rangle$ to results related by $\langle l_2\rangle$, or, using the PER notation from [36, 70]:

$$a : \langle l_1\rangle \Rightarrow \langle l_2\rangle$$

The intensional interpretation makes finer distinctions than the equivalence relation interpretation, and these distinctions are motivated by the requirement to express not only *what* is released, but to provide some control of *how* information is leaked. Take for example the policy consisting of the single function $\lambda x.\lambda y.x == y$. In principle this function can reveal everything about a given secret first argument, via suitably chosen applications. In the extensional interpretation this policy is therefore equivalent to the policy represented by $\lambda x.x$. However, the intensional interpretation of labels distinguishes these policies—the intuition being that it is much harder (slower) to leak information using the first function than using the second.

[4] We focus here on the so-called *local* policies.

In conclusion we see that relaxed noninterference can be understood in terms of PERs under an extensional interpretation, but provides potentially more information with an intensional interpretation. What is missing in this characterisation is a clear semantic motivation for *which* intensional equivalence is appropriate, and what general guarantees it provides. One suggestion[5] is to use a complexity preserving subset of extensional equivalence. This should guarantee that the attacker cannot leak secrets faster than if he literally used the policy functions. However it should be noted that the syntactic equivalence from [42] (which include, amongst other things, call-by-name β-equivalence) is *not* complexity-preserving for the call-by-value computation model used therein. See [72] for an exploration of complexity preservation issues. The intensional view of relaxed noninterference thus requires the addition of information about the speed of computation, something which is covered by the "when" dimension.

Quantitative abstractions Under the category "what" we also include properties which are abstractions of "what." One extreme abstraction is to consider the *quantity* of information released. Thus we consider "how much" to be an abstraction of "what." The most direct representation of this idea is perhaps the information-theoretic approach by Clark et al [11], which aims to express leakage in terms of an upper bound on the number of information-theoretic bits. The approach of Lowe [45] can be thought of as an approximation of this in which we assume the worst-case distribution. With this approximation the measure corresponds, roughly, to counting the number of equivalence classes in an equivalence-relation model. For a framework that integrates attacker belief into the analysis of quantitative information flow in a language-based setting see recent work by Clarkson et al. [14].

2.2 Who

It is essential to specify *who* controls information release in a computing system. Ignoring the issue of control opens up attacks where the attacker "hijacks" release mechanisms to launder secret information. Myers and Liskov's *decentralised label model* [55] offers security labels with explicit ownership information (see, e.g., [24, 61, 6] for further ways of combining information flow and access control). According to this approach, information release of some data is safe if it is performed by the owner who is explicitly recorded in the data security label. This model has been used for enhancing Java with information flow controls [54] and has been implemented in the Jif compiler [58].

The key concern about ownership-based models in general is assurance that information release cannot be abused by attackers. As a step to offer such an assurance, Zdancewic and Myers have proposed *robust declassification* [79] which guarantees that if a passive attacker may not distinguish between two memories where the secret part is altered then no active attacker may distinguish between these memories.

Recent work by Myers et al. [56] connects ownership-based security labels and robust declassification by treating ownership information as *integrity* information in the data security labels. In this interpretation of robust declassification, information release

[5] [Peng Li, personal communication]

is safe whenever no change in the attacker-controlled code may extract additional information about secrets. Interestingly, this implies that declassification annotations of the form `declassify`(e) do not pertain to the "what" (the value of expression e) or the "where" (in the code) dimensions because robust declassification accepts any leak as intended unless the active attacker may affect it.

Furthermore, *qualified robustness* is introduced, which provides the attacker with a limited ability to affect what information may be released by programs. Dually to declassification, an `endorse` primitive is used for upgrading the integrity of data. Once data is endorsed to be trusted, it can be used in decisions on what may be declassified. Qualified robustness intentionally disregards the values of endorsed expressions by considering arbitrary values to be possible outcomes of endorsement.

Tse and Zdancewic also take the decentralised label model as a starting point. They suggest expressing ownership relations via subtyping in a monadic calculus and show that typable programs satisfy two weakened versions of noninterference: *conditioned noninterference* and *certified noninterference* [74].

2.3 Where

Where in a system information is released is an important aspect of information release. By delegating particular parts of the system to release information, one can ensure that no other (potentially untrusted) part can release further information.

Considering *where* information is released, we identify two principal forms of locality:

Level locality policies describing where information may flow relative to the security levels of the system, and

Code locality policies describing where physically in the code information may leak.

The common approach to expressing the level locality policies is *intransitive noninterference* [64, 59, 63, 47]. Recall that confidentiality policies in the absence of information release are often regulated by conventional *noninterference* [30, 77], which means that public output data may not depend on (or interfere with) secret input data. However, noninterference is over-restrictive for programs with intentional information release (average salary, information purchase and password checking programs are flatly rejected by noninterference). Intransitive noninterference is a flow-control mechanism which controls the path of information flow with respect to the various security levels of the system. For standard noninterference the policy is that information may flow from lower to higher security levels in a partial order (usually a lattice) of security levels. The partial order relation between levels $x \leq y$ means that information may freely flow from level x to level y, and that an observer at level y can see information at level x. Since the flow relation $\leq$ is a partial order it is thus always transitive. Intransitive noninterference allows more general flow policies, and in particular flow relations which are not transitive. The canonical example (but not the only use of intransitive noninterference) is the policy that says that information may flow from *low* to *high*, from *high* to a *declassifier* level and from the *declassifier* level to *low*, but *not* directly from *high* to *low*. The definition of intransitive noninterference must

ensure that all the downgraded information indeed passes through the declassifier, and is thereby controlled.

Mantel [47] has introduced a variant of this idea which separates the flow policy into two parts: a standard flow lattice ($\leq$), together an intransitive downgrading relation ($\rightsquigarrow$) for exceptions to the standard flow. This has been adapted to a language-based setting by Mantel and Sands [49], in which both kinds of locality are addressed: intransitive flows at the lattice level, associated to specific downgrading points in the code.

Code locality can be thought of as a simple instance of intransitive noninterference based loosely on the idea of all "leaks" passing through a declassification level. Roughly speaking, the declassification constructs in the code can be thought of as the sole place where information should violate the standard flow policy. Thus the definition of intransitive noninterference should ensure that the only information release in the system passes through the intended declassifications and nothing more. With this simple view, the approaches of Ryan and Schneider (so called *constrained noninterference* [66]), Mullins[6] [53], Bossi et al. [8], and Echahed and Prost [22, 23] could also be seen as forms of intransitive noninterference.

Most recently, Almeida Matos and Boudol [4] define a notion of *non-disclosure*. They introduce an elegant language construct `flow` F `in` M to allow the current flow policy to be extended with flows F during the computation of M. To define the semantics of *non-disclosure* the operational semantics is extended with policy labels. This could be seen as a generalisation of the "default base-policy + downgrading transitions" found in Mantel and Sands' work. Although developed independently, the bisimulation-based definition of security is very close to that of Mantel and Sands, albeit less fine grained and focused purely on code locality.

Three other recent papers describe dynamic policy mechanisms whereby an information flow policy can be dynamically modified by the program.

The first, described by Hicks et al. [35] investigates an information flow type system for a small functional language with dynamic policies based on the decentralised label model [55]. They coin the phrase *noninterference between updates*. This phrase intuitively captures a natural security requirement in the presence of policy changes. However in terms of semantic modelling the only precise semantic condition provided in [35] might be more accurately described as *noninterference in the absence of updates*—which is essentially the principle of *conservativity* described in Section 3.

In the second work [17], Dam describes a policy update mechanism in which the security level of variables can be dynamically reclassified. Security is then defined, like for Mantel and Sands [49], as an adaptation of strong low-bisimulation [69]. Interestingly, Dam's definition does not coincide with that of Mantel and Sands on the common subset of features. Consider the program

$$\texttt{if } h = 0 \texttt{ then } [l := h_1] \texttt{ else } [l := h_2]$$

where $[\ldots]$ denotes a declassified assignment. Unlike the definition of [49] Dam's definition deems this program to be secure. In [49] one would instead have to also

[6]Despite the similarity in terminology, there is no tight relation between Mullins' admissible interference [53] and Dam and Giambiagi's admissibility [18]; in fact the two conditions emphasise different dimensions of declassification.

declassify the result of the boolean test, e.g.,

$$[l_0 := (h = 0)]; \texttt{if } l_0 \texttt{ then } [l := h_1] \texttt{ else } [l := h_2]$$

One might argue that Dam's definition does not exhibit *code locality*—at least not to the extent of [49]; see [17] for an alternative perspective.

In the third approach [32], Gordon and Jeffrey consider dynamically generated security levels in the context of π-calculus. Generalising Abadi's definition of secrecy for spi-calculus [2], they propose a notion of *conditional secrecy*, which guarantees that secrets are protected unless particular principals are compromised.

A combination of "where" and "who" policies in the presence of encryption has been recently investigated by Hicks et al. [34]. The authors argue that declassification via encryption is not harmful as long as the program is, in some sense, noninterfering before and after encryption. In order to accommodate this kind of leak locality property, they define two semantics: operations semantics whose job is to evaluate program parts free of cryptographic functions and reduction semantics for evaluating cryptographic functions. The key policy is *noninterference modulo trusted functions*, which is intended to guarantee that if all encryptions are trusted (a trust relation is built in the underlying security lattice) then the attacker is not able to distinguish secrets through a visibility relation that ignores differences arising from the application of cryptographic functions. They show that if no trusted cryptographic functions are used, their security characterisation reduces to noninterference.

2.4 When

The fourth dimension of declassification is the temporal dimension, pertaining to *when* information is released. We identify three broad classes of temporal release specification:

Time-complexity based Information will not be released until, at the earliest, after a certain time. Time is an asymptotic notion typically relative to the size of the secret.

Examples of this category include Volpano and Smith's relative secrecy and one-way functions [76, 75]. In these cases the security definition says that the attacker cannot learn the (entire) value of a secret of size n in polynomial time. Putting it another way, the secret may be leaked only after non-polynomial time. This is related to the approaches found in Laud's work [40, 41] and Mitchell et al.'s work on polynomial-time process calculus (see, e.g., [44, 52]). Here the attacker is explicitly given only polynomial computational power, and under these assumptions the system satisfies a noninterference property.

Probabilistic With probabilistic considerations one can talk about the probability of a leak being very small. This aspect is also included in Mitchell et al.'s work, and complexity-theoretic, probabilistic and intransitive noninterference are combined in recent work of Backes and Pfitzmann [5]. The notion of approximate noninterference from [21] is more purely probabilistic: a system is secure if the chance of an attacker making distinctions in the values of secrets is smaller than

some constant ϵ. We view this (arguably) as a temporal declassification since it essentially captures the fact that secrets are revealed *infrequently*.

Relative A non-quantitative temporal abstraction involves relating the time at which downgrading may occur to other actions in the system. For example: "downgrading of a software key may occur after confirmation of payment has been received."

The work of Giambiagi and Dam [29] focuses on the correct implementation of security protocols. Here the goal is not to prove a noninterference property of the protocol, but to use the components of the protocol description as a specification of *what* and *when* information may be released. The idea underlying the definition of security in this setting is *admissibility* [18]. Admissibility is based on invariance of the system under systematic permutations of secrets.

Chong and Myers' security policies [9] address *when* information is released. This is achieved by annotating variables with types of the form $\ell_0 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_k}{\rightsquigarrow} \underline{\ell_k}$, which intuitively means that a variable with such an annotation may be subsequently declassified to the levels $\ell_1, \ldots, \ell_k$, and that the conditions $c_1, \ldots, c_k$ will hold at the execution of the corresponding declassification points. An example of a possible domain of conditions is predicates on the variables in the program.

3 Some principles for declassification

In addressing the issue of what constitutes a satisfactory information release policy, it is crucial to adequately represent the attacker model against which the system is protected. A highly desired goal is a declassification framework that allows for modelling the different aspects of attackers, enabling the system designer to tune the level of protection against each of the dimensions. As a first step toward such a framework, we suggest some principles for declassification intended to serve as sanity checks for existing and emerging declassification models.

This section discusses the semantic consistency, conservativity, monotonicity of release, non-occlusion and trailing attack principles. For convenience, a partial mapping of these principles to some of the models from the literature is collected in Table 1. It is not the goal of this section to be complete with respect to all definitions mentioned in Section 2.

3.1 Semantic consistency

This principle has its roots in the full abstraction problem [60, 51], which has important implications for computer security [1]. Full abstraction is about preserving equivalence by translation from one language to another. Viewing equivalence as the attacker's view (cf. Section 2) of the system, *semantic consistency* ensures that the view (and hence the security of the system) is preserved whenever some subprogram c is replaced by a semantically equivalent program d (where neither c nor d contains declassification). Hence, the first principle:

Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
What				
Partial release [16, 38, 70, 27, 28]	✓	✓	N/A	✓
Delimited release [68]	✓	✓	✓	×
Relaxed noninterference [42]	×	✓	✓	✓
Naive release	✓	✓	✓	×
Who				
Robust declassification [56]	✓*	✓	✓	✓
Qualified robust declassification [56]	✓*	✓	✓	×
Where				
Intransitive noninterference [49]	✓*	✓	×	✓
When				
Admissibility [18, 29]	×	✓	×	✓
Noninterference "until" [9]	×	×	✓	✓
Typeless noninterference "until"	✓*	✓	×	×

* Semantic anomalies

Table 1: Checking principles of declassification.

SEMANTIC CONSISTENCY

> The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms.

This principle aids in modular design for secure systems. It allows for independent modification of parts of the system with no information release, as long as these modifications are semantics-preserving. A possible extension of this principle would be one that also allows modification of code with information release, as long as new code does not release more information.

We inspect each of the release dimensions and list some approaches that satisfy this principle (and some that do not).

What Models capturing what is released are generally semantically consistent. Because what is released is described in terms of program semantics, changing subprograms by semantically equivalent ones does not make a difference from the security definition's point of view. This argument applies to partial release [16, 38, 70, 27, 28] and delimited release [68][7]. Relaxed noninterference [42] aims to provide more than just "what" properties, and does so through the use of a decidable notion of equivalence (as discussed in Section 2.1). Thus semantic consistency fails when we transform outside of this relation.

Who The attacker's view in the robust declassification specification [56] is defined by low-level indistinguishability of traces up to high-stuttering (traces must agree on the sequence of assignments to low variables). Assuming that semantic equality implies low-level indistinguishability, the end-to-end nature of robustness ensures that exchanging semantically equivalent subprograms may not affect program security. This argument extends to qualified robustness [56]. This characterisation is insensitive to syntactic variations of subprograms that are free of declassification as long as their semantics are preserved (which, for example, means that reachability is not affected).

Where The language-based intransitive noninterference condition of Mantel and Sands [49] satisfies semantic consistency. The definition of intransitive noninterference is built on top of a notion of k-bisimulation, where k is a security level. The basic idea is that when a declassification step occurs between two levels l and m, then (i) nothing other than that visible at level l is released, and (ii) it is only visible to the observer at level k if $m \leq k$ (i.e., k is authorised to see information at level m). After each step the bisimulation definition (following [69]) requires the program parts of the configurations be again bisimilar *in all states*. This is a form of "policy reset," and the same approach is adopted in *non-disclosure* [4].

Bossi et al's condition [8] is described in an extensional way which strongly suggests that it also satisfies semantic consistency. Echahed and Prost's condition [22, 23] is based on a rather unusual general computational model (a mixture

[7] As elsewhere, we state this without proof (and hence with due reservations), as proofs would require a lot of detail to be given.

of term-rewriting, constraint and concurrent declarative programming) which makes it more difficult to assess.

When The semantic consistency principle critically depends on the underlying semantics. For complexity-sensitive security definitions [76, 75, 40, 41, 44, 52], semantic consistency requires complexity-preserving transformations. Otherwise, for example, a program which cannot leak in polynomial time could be sped up by a transformation that compromises security.

Dam and Giambiagi's admissibility [18, 29] does not satisfy the semantic consistency principle. Due to the syntactic nature of admissibility, it is possible to replace functions in the declassification protocol with semantically equivalent ones so that admissibility is not preserved. Take policy P that only allows leaking secrets via function f (which could be, for example, an encryption function). A program $S = f(h)$ is then admissible with respect to P. However, suppose the semantics of f is the identity function. Changing the program S by a semantically equivalent program $S' = h$ results in a program that is not admissible with respect to P. Notice that this is an instance of a general phenomenon: syntactic definitions of security are bound to violate the semantic robustness property. In the case of admissibility, recent unpublished results [19] introduce semantic information into admissibility policies via *flow automata*. This appears to be a useful feature for recovering semantic consistency.

Chong and Myers' *noninterference "until"* [9] is somewhat different, in that they first define a base security type-system for a mini ML-like language for handling noninterference, and *then* define the intended security condition, but only over terms which are typable according to the base security type system.

No security definition which demands that terms are typed according to a security type system (or any other computable analysis) can satisfy the semantic consistency principle with respect to *all* programs. This follows from a simple computability argument: semantic equivalence is typically not recursively enumerable, but the set of "typable" programs is either recursive or at the very least recursively enumerable. Thus there are pairs of equivalent terms for which one is typable and the other is untypable. An untypable term, according to such a definition, cannot be considered secure.

Definitions which depend on such specific analyses are not entirely satisfactory as semantic definitions. For example, a program such as

$$\texttt{if } h > h \texttt{ then } l := 0$$

would be considered insecure by the definition since the type system makes the usual coarse approximations. In order to recover semantic consistency, the obvious fix is to lift the restriction that the definition applies to well-typed programs. Along with our analysis of the noninterference "until" definition [9], presently we also explore the consequences of this generalised definition, which we refer to as *typeless noninterference "until"*. Most recently, a similar generalisation has been defined by Chong and Myers in order to combine declassification and so-called *erasure policies* [10].

Semantic anomalies The notion of semantic consistency is, of course, dependent on the underlying semantic model. The base-line semantic model can be thought of as defining the attacker's intrinsic observational ability. In many cases the base-line semantics is not given explicitly. However, the definition of security is often built from a notion of program equivalence which takes into account security levels. In such cases it is natural to induce the "base-line" semantics—the attacker's observational power—by considering the notion of equivalence obtained by assuming the degenerate one-point security lattice.

Having done this we can observe whether the induced semantics is a "standard" one. In the case that it is nonstandard we call this a *semantic anomaly*, which reflects something about our implicit attacker model. The presence of semantic anomalies means that semantic consistency only holds for that specific semantics.

We have noted that the complexity-based definitions naturally require that the semantic model which preserves complexity—i.e., that the notion of equivalence must take into account computational complexity. This is perhaps not standard, but rather natural in this setting. We point out several examples of clear semantic anomalies, coming from Myers et al.'s robust declassification [56], Mantel and Sands' version of intransitive noninterference [49], Almeida Matos and Boudol's non-disclosure property [4] and the typeless variant of noninterference "until." Each of these has an attacker model which turns out to be stronger than strictly necessary. Although it is safe to assume that an attacker has greater powers than he actually possesses, it has a potential disadvantage of rejecting useful and intuitively secure programs. They only satisfy semantic consistency under a stronger than usual semantic equivalence: this equivalence is extracted from the underlying indistinguishability relations by viewing all data as low.

- In the case of robust declassification [56] the semantic model allows the attacker to observe the sequence of low assignments (up to stuttering). This means that for any low variable l, the command $l := l$ is considered semantically distinct from `skip`, since the former contains a low assignment and the latter does not. Under more standard semantics, a transformation from `if` h `then skip else skip` to `if` h `then` $l := l$ `else skip` (where h is high and l is low) would be semantics-preserving. However, the first program satisfies robust declassification whereas the second one does not, which would break semantic consistency. Similar transformation can be constructed for the following items.

- In the case of intransitive noninterference [49], the semantic model is explicitly given as a strong bisimulation, so that only computations which proceed at the same speed are considered equivalent. It is argued, with reference to [69], that this allows for useful attacker models in which the attacker controls the thread scheduler. However, with a coarser base-line model it is not immediately clear what an appropriate definition of intransitive noninterference should be.

- The implicit semantics of Almeida Matos and Boudol [4] is arguably most anomalous of those considered here. This is due to the fact that their language contains local variables. These do not fit well with the stateless bisimulation-based notion of equivalence that is induced. Consider the following example, using the usual

two-level base policy:

$$\begin{array}{l}\texttt{let}\ u_L = \texttt{ref false in}\\ \quad \texttt{if}\ !u_L\ \texttt{then}\ v_L := w_H\end{array}$$

where $!u_L$ returns the value that the reference u_L points to. This program is considered insecure, because the semantics induced by the notion of bisimulation assumes that the insecure command $v_L := w_H$ is reachable, even though it is clearly unreachable in any context. Here the anomaly cannot be described in terms of what the attacker can *observe*, but rather as an implicit assumption that the attacker can even *modify* the value of local variables. This is clearly stronger than necessary.

- For typeless noninterference "until," there are technical issues with the definition of noninterference which assumes that an attacker can distinguish what would be normally considered equivalent functions. As a result, the program below is considered insecure:

$$\texttt{if}\ x_h\ \texttt{then ref}(\lambda x.x+1)\ \texttt{else ref}(\lambda x.1+x)$$

This could be seen as a failure of semantic consistency, since transforming $1+x$ into $x+1$ would make the program secure. However, we view this problem as a minor technical artifact that could be fixed for example by only allowing the attacker to observe stored values of ground type.

3.2 Monotonicity of security

Declassification effectively creates a "hole" in the security policy. For a given security definition, a program such as $l := h$ might well be considered insecure, but by adding a declassification annotation to get $l := \texttt{declassify}(h)$ the program may well be considered secure. So the natural starting point—the base-line policy—is that absence of declassification in a program or policy implies that the program should have no insecure information flows. On the other hand, the more declassification annotations that a program contains, the weaker the overall security guarantee.

This leads to two related principles: *conservativity*, which says that security conservatively extends the notion of security for a language without declassification, and *monotonicity of release* based on the monotonicity of security with respect to increase in declassification annotations in code (the more declassifications you have the more "secure" you become). Let us consider each of these principles in turn.

It is sensible to require that programs with no declassification annotations or declarations satisfy a standard security property that allows no secret leaks, as commonly expressed by some form of noninterference [30]. As before, we use the term noninterference to refer to the standard zero information flow policy for the language. We arrive at a principle that requires declassification policies to be conservative extensions of noninterference:

CONSERVATIVITY

Security for programs with no declassification is equivalent to noninterference.

Notice that this principle is straightforward to enforce by making it a part of security definition, which would have the flavour of "a program is secure if either it is noninterfering or it contains declassification and satisfies some information release policy." Often the conservativity principle holds trivially as it is built directly from a definition of noninterference.

Nevertheless, noninterference "until" [9], in the case when there is no declassification in either the policy or code, yields a strict (decidable) subset of noninterference. So, taking the same example as previously, `if` $h > h$ `then` $l := 0$ is considered insecure because the type system rejects it. Thus the definition does not strictly satisfy conservativity.

Many mechanisms for declassification employ annotations to the code (or some other specification) which denote where a declassification is intended. Operationally these declassification annotations do not interfere with normal computation. At the level of annotations, the more declassify annotations in a program, the weaker the overall security guarantee. This common-sense reasoning justifies the *monotonicity of release* principle:

MONOTONICITY OF RELEASE

Adding further declassifications to a secure program cannot render it insecure.

or, equivalently, an insecure program cannot be made secure by *removing* declassification annotations.

We now revisit the dimensions of release and apply the monotonicity of release principle to security characterisations along the different classes.

What Most of the examples in this class (that we have considered) express policies extensionally, so they do not rely on annotations to define the semantics of declassification. One exception is delimited release, which uses the collection of annotations in the code to determine the global policy. Adding an annotation gives the attacker more knowledge, so there is less remaining to attack. Thus if a system is secure with respect to a given degree of attacker knowledge, adding more knowledge will never make it insecure. In the PER interpretation this is just a standard monotonicity property: if a system, when presented with any two states related by some binary relation R produces equivalent observable outputs, then the same will be true when replacing R by any S such that $S \subseteq R$.

Who Declassification annotations are not used by the semantic definition of robustness or qualified robustness [56] (although these annotation are used in the static analysis). Therefore, program security is invariant under the removal or addition of declassification annotations. The situation is different with the removal or addition of *endorsement* annotations for qualified robustness, however [56]. Endorsement statements have a *scrambling* semantic interpretation that allows for arbitrary values to be the outcome of endorse. Inspired by "havoc" commands [38], this semantic treatment allows the difference between two values of a variable to be "forgotten" by forcing this variable to take an arbitrary

value. This is justified when each endorse is preceded by a placeholder for attacker-controlled code [57]. In this case, arbitrary values may be set to attacker-controlled variables when the control reaches the endorse. However, in general the scrambling interpretation of endorse (or declassify) might lead to the reachability of code that is otherwise dead. Dead code may mask security flaws, as we will see below.

It is not necessary to introduce endorsement in order to explain these reachability issues in security definitions. Consider two types of declassification, by scrambling the source and target of declassification, respectively. The intention is to "forget" the effect of declassification by requiring the source h (or target l) of a declassification operation $l := \texttt{declassify}(h)$ to take any possible value. Under the former, the semantic treatment of declassification is specified nondeterministically by

$$\langle l := \texttt{declassify}(h), s\rangle \longrightarrow s[h \mapsto val, l \mapsto val]$$

for all val, which corresponds to scrambling the values of both h and l with value val. Under the latter, the scrambling is done at the result of declassification:

$$\langle l := \texttt{declassify}(h), s\rangle \longrightarrow s[l \mapsto val]$$

which does not affect the value of h but makes any value of l a possible outcome of declassification. With the respective semantic interpretations of declassify, the security condition is *possibilistic* noninterference, requiring the indistinguishability of possibilities for low-level output as high-level input is varied. We now see why both of these interpretations break the monotonicity of release. Consider the program:

$$\begin{array}{l} h := 0; \\ l := \texttt{declassify}(h); \\ \texttt{if } l = 42 \texttt{ then } l' := h' \end{array}$$

The program is clearly secure if declassification is removed. In the presence of declassification, however, the value of l might become 42 under both scrambling semantics, and thus the insecure code $l' := h'$ becomes reachable. Hence, the program is insecure under both semantics, which would contradict the monotonicity of release.

Where The annotations of [49] apply to simple assignment statements. From a local perspective these would seem to satisfy the monotonicity principle, since the conditions required for a normal assignment statement are strictly stronger than for a downgrading statement. However, the fly in the ointment, as far as monotonicity is concerned, is that the definition effectively assumes that the attacker can observe the fact that a declassification operation is being performed, regardless of its content. Thus a program such as

$$\texttt{if } h = 42 \texttt{ then } [l := l] \texttt{ else } l := l$$

where, as before, $[\ldots]$ denotes a declassified assignment, is insecure, because an attacker observing the presence or absence of a declassification action learns

whether h was 42 or not. Removing the declassification makes the program secure.

However, in this case the fix to the definition from [49] seems straightforward: when nothing is leaked by the declassification then it can be viewed as a non-declassification, and vice-versa.

When Complexity-based and probabilistic declassifications are expressed extensionally, so the monotonicity principle does not apply.

For other temporal declassification conditions there is potential for monotonicity of release to fail for general reasons. Programs may contain declassifications which are in fact harmless—i.e., they do not violate noninterference, such as the declassification of a known constant. But if the policy refers directly to the presence or absence of the declassification operation itself, then the very fact that a declassification statement is present in the code—albeit harmless—may cause it to violate the policy. Removing the declassify annotation might, by the same token, cause the policy to be satisfied.

By this argument, it is possible to show that admissibility [18, 29] does not satisfy the monotonicity of release principle. The semantics of the protocol that specifies declassification is abstracted away, enabling harmless declassification to be disguised by the protocol's syntactic representation. Similar to semantic consistency, monotonicity of release is likely to be recovered for admissibility by introducing semantic information about declassification via flow automata [19].

Typeless noninterference "until" also fails monotonicity of release. One example, following the general reasoning above, is when the conditions used to trigger declassifications refer to declassifications themselves. A natural example would be a policy that says that A can be declassified if B has *not* been declassified, and vice-versa. This ensures that at most one of A and B are declassified—an interesting policy if A is the one-time pad and B is the encrypted secret. Now suppose that a program declassifies A, but contains a harmless declassification for B (i.e., one that does not *actually* reveal anything about B). The program is insecure according to the policy, but if the declassification is removed from the harmless release of B, the program becomes secure. Note that monotonicity of release is preserved by noninterference "until" [9] because declassification conditions are simply assumed to always be guaranteed by typed programs.

3.3 Non-occlusion

Whenever declassification is possible, there is a risk of laundering secrets not intended for declassification. Laundering is possible when, for example, declassification intended only for encrypted data is applied to high plaintext. This is an instance of *occlusion*. A principle that rules out occlusion can be informally stated as follows:

NON-OCCLUSION

The presence of a declassification operation cannot mask other covert information leaks.

The absence of occlusion is a useful property of information release. Generally, declassification models along the "what" dimension satisfy this principle. Occlusion is avoided because covert flows, by the spirit of "what" models, should not increase the effect of legitimate declassification. An exception is the delimited release policy where globally declassified expressions are extracted from everywhere in the code. Recall the example:

$$\texttt{if true then } l := h * h \texttt{ else } l := \texttt{declassify}(h * h)$$

Unreachable declassification in the `else` branch covers up the leak in the `then` branch.

Occlusion is prevented in robust declassification [56] because declassification annotations in code may not affect robustness (and hence all declassification is considered intended as long as the attacker may not affect it). Also, intransitive noninterference [49] successfully avoids occlusion by resetting the state (and thus the effect of each declassification), as a consequence of small-step compositional semantics.

However, qualified robustness [56] and noninterference "until" [9] (when typability of programs is not required) are both subject to occlusion.

Occlusion in qualified robustness [56] can be illustrated by occlusion in the source- and target-scrambling release definitions from Section 3.2 (examples for the actual qualified robustness definition can be found in [57]). Indeed, slightly modifying the example we receive the program:

$$\begin{array}{l} h := 0; \\ l := \texttt{declassify}(h); \\ \texttt{if } l = 42 \texttt{ then } l' := \texttt{declassify}(h'); \\ l' := h' \end{array}$$

The program is insecure because it leaks h' into l', and because $l' := \texttt{declassify}(h')$ is not reachable. However, under the scrambling semantics, the value of l might become 42 after the first declassification, implying possible reachability of the second declassification in dead code which masks the real leak $l' := h'$. Indeed, any value is a possible final value for l' in the above program regardless of the initial value for h'. Hence the program is deemed secure with respect to both source- and target-scrambling definitions.

A generalisation of security policies "until" [9] to all programs (typeless noninterference "until") leads to occlusion. Because the security condition only considers $c_1 \ldots c_k$-free traces, i.e., traces that have not reached the last declared declassification, it is insensitive to injections of harmful code with unintended leaks after the last declassification. For instance, consider a password checking example from [9]:

$$\begin{array}{l} \textbf{int}_H\, secret := \ldots; \\ \textbf{int}_H\, pwd := \ldots; \\ \textbf{int}_H\, guess := getUserInput(); \\ \textbf{boolean}_H\, test := (guess == pwd); \\ \textbf{boolean}_L\, result := \texttt{declassify}(test, H \rightsquigarrow \underline{L}); \\ \ldots \end{array}$$

and inject code that leaks $secret$ at level H after the password is checked. This leak is not prevented despite the fact that the security condition is variable-specific (i.e.,

it states noninterference "until" for each variable separately). Note that the original version of noninterference "until" [9] prevents such attacks by considering untyped programs as insecure from the outset. The two programs above inspire the following general principle (which can be viewed as an instance of non-occlusion):

TRAILING ATTACKS

Appending an insecure program with fresh variables to a secure terminating program should not result in a secure program.

One way of protecting against trailing attacks could be to introduce a special level ℓ_T that corresponds to termination and require that any declassified data be declassified to ℓ_T at the end of overall computation.

Consider a *naive release* policy that states that a program is secure if for any two runs either they preserve low equivalence of traces, or one of them executes a declassify statement. Clearly, this policy satisfies semantic consistency as semantics-preserving transformation of subprograms without declassification may not affect indistinguishability of traces for low-level observer. It also satisfies conservativity, by definition. Further, monotonicity of release also holds because the removal of declassification from an insecure program (there must exist a pair of traces without declassification that are not low-indistinguishable) may only result in an insecure program (by the same pair of traces). Although these principles hold, they are not sufficient for security assurance. This is reflected by occlusion, which naive release suffers from.

A final remark on the principles is that they are not intended to be universally necessary for all declassification policies. For example, sometimes policies are of syntactic nature by choice (as, for instance, admissibility), which makes it infeasible to guarantee appealing semantic principles. However, we argue that if one of the principles fails, it is an indication of a potential vulnerability that calls for particular attention as to why the principle can be relaxed.

4 Conclusion

Seeking to enhance understanding of declassification, we have provided a road map to the area of information release. The classification of declassification policies according to "what," "who," "where" and "when" dimensions has helped clarify connections between existing models, including the cases when these connections were not, in our opinion, made entirely accurate in the literature. For example, abstract noninterference [27, 28] and relaxed noninterference [42] fall under "what" models in our classification, which disagrees with connections to "who" and "where" definitions made in the respective original work. Another example is the deceptively akin admissibility [18] and admissible interference [53] that address different dimensions of declassification ("when" and "where").

A reasonable question is to what extent the dimensions can be made formally precise. For a given model, the "what" and "when" dimensions seem relatively straightforward to define formally. The "what" dimension abstracts the extensional semantics of the system; the "when" dimension can be distinguished from this since it requires an

intensional semantics that (also) models time, either abstractly in terms of complexity or via intermediate events in a computation. The "who" and "where" dimensions are harder to formalise in a general way, beyond saying that they cannot be captured by the "what" and "when" dimensions.

Why not "how" or "why"? It is natural to ask whether the interrogative words "how" and "why" are also reasonable dimensions. Regarding specifications of "how" information is released, we argue that these are covered by some combination of "what" and "when" and "where". For example, suppose that some particular functions may be used for declassification. With an intensional view this can be modelled simply in the "where" dimension (code locality). At the other extreme we may want a more extensional view, in which case we are in the purely "what" dimension. In between we have many possibilities using the "when" dimension to capture the speed of release (the essence of the algorithm) or some other temporal constraints.

Why is information released? It is clearly important to know the reason behind the intentional release of information. But this dimension seems inappropriate to consider at the code level since it deals with issues that come before the coding phase. In general the "why" dimension is application-specific, and its study is more naturally part of the software design and development process. See, e.g., [33] for one of the few works that aim to integrate security specification and declassification (via the decentralised label model) into language-based security.

Dimensions for integrity From the point of view of information flow, integrity is often seen as a natural dual of confidentiality. One wishes to prevent information flows from untrusted (low integrity) data sources to trusted data or events. As for confidentiality, there are many situations where we might wish to upgrade the integrity levels of data (so-called *endorsement*). It seems that the dimensions for declassification could also be applied to endorsement:

What The "what" dimension can be studied with essentially the same semantics, and thus deals with what parts of information are endorsed. Interestingly, Li and Zdancewic [43] in a study of the dualisation of relaxed noninterference [42], discuss some non-dual aspects of policies, stemming from whether the code itself is trusted or not.

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

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

Who The "who" dimension is interesting because the notion already embodies a form of integrity. Robust declassification, for example, argues that low integrity data should not effect the decision of what gets declassified [56]. For integrity we might thus define a notion of *robust endorsement* to mean that the decision to endorse data should not itself be influenced by low integrity data. This approach can benefit from a non-dual treatment of endorsement. Because the potentially dangerous operations like declassification and endorsement are "privileged" operations, it might make sense to apply *similar*, not *dual* constrains.

Challenges for enforcement mechanisms This paper is concerned with policies for information release. While a comprehensive treatment of declassification policy enforcement is outside the scope of this paper, we briefly highlight some challenges for enforcing policies along each of the dimensions.

What Enforcement of "what" policies can be delicate, which we demonstrate by examples of delimited release and information-theoretic policies. In delimited release, declassification policies are expressed in code by $\texttt{declassify}(e, \ell)$ annotations. Programmers may use these annotations in order to declare that the value of expression e can be released to level ℓ, naturally suggesting what is released. However, care must be taken in order for the declaration not to open up possibilities for leaks unrelated to e. Indeed, the value e may change as the program is executed (and can be affected by different secrets—either explicitly or via control flow). An enforcement mechanism that ignores what secrets can affect e over the course of computation might be open to laundering attacks. Preventing secret variables in escape hatches e from depending on other data is a possibility [68] although more permissive enforcement mechanisms (which, in general, might require human assistance) are possible (cf. [7, 20, 73]).

Information-theoretic release policies are notoriously hard to enforce. Tracking the quantity of information through loops is particularly challenging. Known approaches are only able to handle rudimentary loops [13].

When Enforcement of "when" policies can be particularly challenging. Such a mechanism needs to be able to verify that release may only take place when some condition c is satisfied, which often requires temporal reasoning about program behaviour. Furthermore, it is important that no leaks are introduced through dependencies from secrets to condition c itself.

Where Enforcement of "where" policies is more natural. An occurrence of a declassification annotation $\texttt{declassify}(\texttt{e}, \ell)$ reflects both policy locality and code locality. Policy locality is represented by label ℓ, localising the destination of information release. Code locality is represented by where in code the annotation occurs.

Who The decentralised label model [55] is an example of a release mechanism with explicit ownership information. Declassification $\texttt{declassify}(\texttt{e}, \ell)$ is allowed if the level of e is downgraded in a way that only affects the owner of the code

that runs the declassification command. While this is an intuitive enforcement mechanism, only recently have there been attempts to connect this mechanisms to semantic goals [74].

While the Jif compiler [58] provides support for declassification, its enforcement mechanism is only concerned with the "who" dimension of declassification via ownership in the decentralised label model [55]. In general, it is crucial that enforcement mechanisms are capable of handling release policies along each of the dimensions.

Summing up This paper is a step toward the goal of developing policies that allow combinations of policies from the individual dimensions into solid *policy perimeter defence*. Perimeter defence is a standard principle of network security: as systems are no more secure than their weakest points, they must be defended across the entire perimeter of the network. The ambition with policy perimeter is to prevent attackers from penetrating systems via weakly defended dimensions of information release.

For this to be possible we must have a better understanding of the implications of our security definitions—even more so in the presence of declassification. We have suggested some prudent principles of declassification that further help to avoid vulnerabilities in release policies, and provide tools for better understanding declassification definitions. Measuring our own previous work on declassification against these principles has revealed anomalies and "artifacts" that had previously gone unnoticed, and we suggest that the principles should serve as useful "sanity checks" for emerging models.

Acknowledgements

Thanks are due to Andrew C. Myers and Pablo Giambiagi for fruitful discussions. The paper has also benefited from the comments of Gerard Boudol, Stephen Chong, Mads Dam, Roberto Giacobazzi, Sebastian Hunt, Peng Li, Isabella Mastroeni, David Naumann and the anonymous reviewers.

This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905, and by National project funding from VINNOVA (Swedish Governmental Agency for Innovation Systems), SSF (Swedish Foundation for Strategic Research) and VR (Swedish Research Council).

References

[1] M. Abadi. Protection in programming-language translations. In *Proc. International Colloquium on Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 868–883. Springer-Verlag, July 1998.

[2] M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, September 1999.

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

[4] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240, June 2005.

[5] M. Backes and B. Pfitzmann. Intransitive non-interference for cryptographic purposes. In *Proc. IEEE Symp. on Security and Privacy*, pages 140–153, May 2003.

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

[7] 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.

[8] A. Bossi, C. Piazza, and S. Rossi. Modelling downgrading in information flow security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 187–201, June 2004.

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

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

[11] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

[12] D. Clark, S. Hunt, and P. Malacaria. Non-interference for weak observers. In *Proc. Programming Language Interference and Dependence (PLID)*, August 2004.

[13] D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a While language. In *QAPL'04, Proc. Quantitative Aspects of Programming Languages*, volume 112, pages 149–166, January 2005.

[14] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 31–45, June 2005.

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

[16] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[17] M. Dam. Decidability and proof systems for language-based noninterference relations. In *Proc. ACM Symp. on Principles of Programming Languages*, 2006.

[18] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. IEEE Computer Security Foundations Workshop*, pages 233–244, July 2000.

[19] M. Dam and P. Giambiagi. Information flow control for cryptographic applets. Presentation at the Dagstuhl Seminar on Language-Based Security, October 2003. `www.dagstuhl.de/03411/Materials/`.

[20] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005. (A preliminary version appeared in WITS'03).

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

[22] R. Echahed and F. Prost. Handling declared information leakage. In *Proc. Workshop on Issues in the Theory of Security*, January 2005.

[23] R. Echahed and F. Prost. Security policy in a declarative style. In *ACM International Conference on Principles and Practice of Declarative Programming*, pages 153–163, July 2005.

[24] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 130–140, May 1997.

[25] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. Computer Security*, 3(1):5–33, 1995.

[26] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *Proc. Foundations of Software Science and Computation Structure*, volume 3441 of *LNCS*, pages 299–315. Springer-Verlag, April 2005.

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

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

[29] P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.

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

[31] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, April 1984.

[32] A. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. CONCUR'05*, number 3653 in LNCS, pages 186–201. Springer-Verlag, August 2005.

[33] R. Heldal and F. Hultin. Bridging model-based and language-based security. In *Proc. European Symp. on Research in Computer Security*, volume 2808 of *LNCS*, pages 235–252. Springer-Verlag, October 2003.

[34] B. Hicks, D. King, and P. McDaniel. Declassification with cryptographic functions in a security-typed language. Technical Report NAS-TR-0004-2005, Network and Security Center, Department of Computer Science, Pennsylvania State University, May 2005.

[35] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Workshop on Foundations of Computer Security*, pages 7–18, June 2005.

[36] L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.

[37] Sebastian Hunt and Isabella Mastroeni. The per model of abstract non-interference. In R. Giacobazzi, editor, *Proc. Static Analysis, 12th International Symposium (SAS'05)*, volume 3184 of *LNCS*. Springer-Verlag, 2005.

[38] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

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

[40] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 77–91. Springer-Verlag, April 2001.

[41] P. Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 159–173. Springer-Verlag, April 2003.

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

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

[44] P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 112–121, November 1998.

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

[46] H. Mantel. Possibilistic definitions of security – An assembly kit –. In *Proc. IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.

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

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

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

[50] I. Mastroeni. On the role of abstract non-interference in language-based security. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3780 of *LNCS*. Springer-Verlag, November 2005.

[51] J. C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 212:141–163, 1993.

[52] J. C. Mitchell. Probabilistic polynomial-time process calculus and security protocol analysis. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 23–29. Springer-Verlag, April 2001.

[53] J. Mullins. Non-deterministic admissible interference. *J. of Universal Computer Science*, 6(11):1054–1070, 2000.

[54] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

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

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

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

[58] 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 2001–2004.

[59] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.

[60] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(1):223–255, December 1977.

[61] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. ACM International Conference on Functional Programming*, pages 46–57, September 2000.

[62] F. Prost. On the semantics of non-interference type-based analyses. In *JFLA'001, Journées Francophones des Langages Applicatifs*, January 2001.

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

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

[65] P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.

[66] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 214–227, June 1999.

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

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

[69] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

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

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

[72] D. Sands, J. Gustavsson, and A. Moran. Lambda calculi and linear speedups. In *The essence of computation: complexity, analysis, transformation*, volume 2566 of *LNCS*, pages 60–82. Springer-Verlag, 2002.

[73] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proc. Symp. on Static Analysis*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, September 2005.

[74] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 279–294. Springer-Verlag, April 2005.

[75] D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.

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

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

[78] S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, August 2004.

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

21st IEEE Computer Security Foundations Symposium

Tractable Enforcement of Declassification Policies

Gilles Barthe
INRIA Sophia Antipolis
Gilles.Barthe@inria.fr

Salvador Cavadini
INRIA Sophia Antipolis
Salvador.Cavadini@sophia.inria.fr

Tamara Rezk
INRIA Sophia Antipolis and MSR-INRIA Joint Centre
Tamara.Rezk@inria.fr

Abstract

Formalizing appropriate information policies that authorize some controlled form of information release, and providing sound analyses for these policies is a necessary step towards practical applications of language-based security.

We propose a modular method to enhance non-interference type systems to support controlled forms of information release that combine the what *and* where *dimensions of declassification. As a case study, we derive from earlier work on non-interference type systems new type systems that soundly enforce declassification policies for sequential fragments of the Java Virtual Machine.*

Our work provides the first modular method to define sound type systems for declassification policies, and the first instance of a sound type system that supports declassification policies for unstructured languages.

1. Introduction

Non-interference [18] is a baseline information flow policy which ensures that publicly available information does not reveal any information about sensitive data during program execution. Non-interference policies are appealing because of the strong security guarantees they provide (zero information leakage); however, practical confidentiality policies cannot be construed to non-interference, as they almost invariably require some constrained release of information.

An important challenge in information flow language-based security is to capture more accurately the kind of confidentiality policies that are needed in practise and to provide tractable and sound enforcement mechanisms for these policies [37]. Declassification policies [35] are weaker forms of information flow policies that permit some constrained information release along four axes regarding *what* specific information is released, *where* within a specific code fragment it is released, *who* releases it, and *when* it is released.

Our contribution is in the line of designing tractable and sound enforcement mechanisms for declassification policies. More concretely, we provide a modular method for achieving sound type systems for declassification from sound type systems for non-interference, and instantiate our method to a sequential fragment of the Java Virtual Machine.

We consider a declassification policy that combines the *what* and *where* dimensions, called *delimited non-disclosure*, that is closely related to the *non-disclosure* policy proposed by Almeida Matos and Boudol [2] and *localised delimited release* proposed by Askarov and Sabelfeld [4]. Informally, delimited non-disclosure is intended to combine two dimensions in a single construct of the form

$$\mathsf{declassify}\ e : \tau\ \mathsf{in}\ c$$

where e is an expression, τ is a security level, and c is a statement.

The choice of the delimited non-disclosure policy is motivated by modularity; our aim is not to propose a new declassification policy but rather to propose a systematic extension of non-interference soundness results in order to achieve declassification soundness for extended non-interference type systems. This modularity is the most distinctive feature of our work. Indeed, existing proofs of soundness proceed by induction over typing derivations, and reproduce a large part of the proof of soundness of the declassification-free fragment of the type system. Instead, we take advantage of the fact that delimited non-disclosure coincides with non-interference for programs without declassification to build a modular proof of soundness. More precisely, we show that if the type system enforces non-interference on the declassification-free fragment of the

978-0-7695-3182-3/08 $25.00 © 2008 IEEE
DOI 10.1109/CSF.2008.11

language, then it enforces delimited non-disclosure. Our method is based on a successor relation between program points, rather than on the syntactic structure of programs, and relies on the idea of control dependence regions, which over-approximate the scope of branching instructions, i.e. instructions that have two or more successors and that can thus yield implicit flows. Here we adopt a local view of policies, which substantially leverages the applicability of information flow type checking and supports more permissive policies.

A second distinctive feature of our approach is its application to low-level languages. In the context of mobile code security, it is essential for code consumers to be able to verify security policies independently and efficiently on the code they receive. Since mobile code applications are typically downloaded in the form of bytecode programs, it is required that verification operates at this level. However, a large body of existing work on language-based security focuses on source languages, and in fact we are not aware of any sound type system that supports declassification policies for unstructured languages. In fact, existing information flow type systems for unstructured languages typically enforce non-interference, see e.g. [6].

Contributions Our main technical contributions are:

- a modular method to define and prove soundness of declassification type systems;
- a case study in which we define and prove soundness of a type system that enforces delimited non-disclosure for a fragment of the Java Virtual Machine;
- a proof of preservation of typing between a type system that enforces delimited non-disclosure in a high-level language and our type system for the JVM.

Organisation The remaining of the paper is organized as follows. Section 2 discusses closely related work. The delimited non-disclosure policy is introduced in Section 3, and Section 4 provides through examples of programs a more detailed comparison with other declassification policies. Section 5 shows how to construct a sound type system from delimited non-disclosure, starting from a type system for non-interference. In Section 6, we instantiate our method to a sequential fragment of the Java Virtual Machine. Section 7 discusses extensions of our work. We conclude in Section 8.

2 Related work

Sabelfeld and Sands [35] analyse main trends on information declassification, and provide a set of principles to be used as "sanity checks" for declassification models. Here we only focus on very closely related work and we refer to Sabelfeld and Sands' survey for a wider overview.

Intransitive non-interference In *intransitive non-interference* policies, flows from high variables to low ones are mediated by a declassifier (or a downgrader), this way non-interference becomes intransitive [30].

In [26], Mantel and Sands introduce an intransitive non-interference based approach concerned to *where* information is declassified.

In this approach, flow ordering is defined by two components: the lattice of security levels, and an "exceptions" relation on security levels expressing special flow cases.

The use of this special relation is deliberately circumscribed to *declassification assignments*, that is, assignments that are allowed to violate the flow ordering (defined by the lattice structure of security levels) as long they respect the "exceptions" relation. This way, information declassification can take place only at specific program points, i.e. at declassification assignments.

Mantel and Sands also present a bisimulation-based definition of their security policy along with a type system to enforce it in a multi-threaded while language.

Localised delimited release Sabelfeld and Myers [34] present *delimited release*, a security policy to declassify information through the special language operator `declassify(e)` where e is globally considered as declassified (*what* dimension).

Later, Askarov and Sabelfeld [4] extended *delimited release* into *localized delimited release*, a security definition combining *what* and *where* dimensions of information declassification. This is made through language semantics instrumentation with the set of declassified expressions and capturing the scope of `declassify(e)` operators.

Non-disclosure Almeida Matos and Boudol [2] proposed the *non-disclosure policy*, a generalisation of non-interference that supports locally induced flow policies through the use of the special construct `flow` $(\mathtt{A} \prec \mathtt{B})$ `in` `S`, where S is a language expression into which flows from principal A to principal B are authorised. Non-disclosure is classified as a declassification policy of the *where* dimension.

Flow locks Broberg and Sands [12] introduce the notion of *flow lock* to specify local information flow policies. In this approach, each variable has attached the set of system principals (or security levels) that can read it. For each principal in this set it is possible to define conditions under which a principal has the right to access to the variable's

value. These circumstances are represented as locks. Local changes of the global policy are specified by means of special program instructions to open and close locks.

Generalisations of non-interference such as non-disclosure and some forms of intransitive non-interference can be represented by flow locks.

Information erasure Hunt and Sands [21] study the semantics of information erasure, a policy aimed at providing guarantees that certain information is not retained after its intended use. Although it is possible to define erasure policies that cover the what and where dimensions of declassification, Hunt and Sands focus on the where dimension. They show that every erasure property can be encoded as (flow-sensitive) non-interference and provide a language construct to express erasure policies in code blocks. They also provide a *global erasure property* and show that it can be enforced by combinning global non-interference and local (command level) non-interference; and define a type system for enforcing the erasure property in a simple while language with inputs and outputs.

The work of Hunt and Sands is inspired by earlier work by Chong and Myers [13], who considered erasure in combination with declassification. Recently, Chong and Myers [14] proposed an information flow type system that enforces non-interference according to a generalization of non-interference introduced by the authors in [13].

Declassification by logic Banerjee et al. [5] present a powerful way to specify security policies including conditions under which declassification is permitted. The specification mechanism is based on *flowtriples*, a combination of program specification and security typing.

The authors also describe an enforcement mechanism integrating security typing (for declassification-free segments of the program), and relational verification and assertion verification (for instructions sequences in flow triples).

$WHERE$, $WHAT_1$ **and** $WHAT_2$ In [24], Mantel and Reinhard propose three security conditions for controlling the *where* and *what* dimensions of declassification. The $WHERE$ condition is similar to intransitive non-interference but it satisfies *monotonicity of release* along with the other three declassification principles from [35]. While both $WHAT_1$ and $WHAT_2$ are applicable to concurrent programs, the former is compositional but does not satisfy *monotonicity of release* principle, and the later satisfies this principle but is not compositional.

The authors provide a type system to enforce *where* and *what* dimensions of declassification in such a way that *all* declassified expressions (what) are allowed to flow to low variables at declassification assignments (where).

3. Delimited Non-Disclosure

In this section, we introduce delimited non-disclosure (DND). We formulate our policy in an abstract setting that can be instantiated to different programming languages.

Program setting We let P range over programs of a given programming language. Each program P has an associated set $\mathcal{P}$ of program points that includes a distinguished entry point entry and a set $\mathcal{P}_{\text{exit}}$ of exit points; we let $i, j, i' \ldots$ range over program points. We assume that for a program P there is a successor relation $\mapsto$ between program points. We let $\mapsto^*$ be the reflexive and transitive closure of $\mapsto$, and assume that, whenever $i \in \mathcal{P}_{\text{exit}}$, there is no successor program point such that $i \mapsto j$. We let $\mathcal{P}^\sharp$ denote the set of branching program points, i.e. those program points that have at least two distinct successors.

One primary target of our work are unstructured programming languages, and therefore we rely on control dependence regions to approximate the scope of branching statements; such control dependence regions have been introduced in the context of non-interference in [22], and used in our earlier work [6]. Formally, we assume given two functions:

$$\begin{array}{lcl} \mathsf{region} & : & \mathcal{P}^\sharp \mapsto \wp(\mathcal{P}) \\ \mathsf{junction} & : & \mathcal{P}^\sharp \rightharpoonup \mathcal{P} \end{array}$$

that respectively provide for each branching program point the set of program points, called region, that execute depending on the branching point; and a junction point (if it exists), that denotes the unique exit point from the region. We assume that these functions correctly over-approximate the scope of branching statements, as formulated in the hypothesis below.

Hypothesis 1 (Exit through junction)**.** *If* $j \in \mathsf{region}(i)$ *and* $k \in \mathcal{P}$ *and* $j \mapsto k$*, then either* $k \in \mathsf{region}(i)$ *or* $k = \mathsf{junction}(i)$.

Furthermore, we assume that the junction point of a region is undefined if the region contains an exit point.

Hypothesis 2 (No return before junction)**.** *If* $j \in \mathsf{region}(i) \cup \mathcal{P}_{\text{exit}}$ *then* $\mathsf{junction}(i)$ *is undefined.*

These hypotheses exactly correspond to the safe over approximation properties **SOAP2** and **SOAP3** introduced e.g. in [6] and used in the case study of Section 6.

The semantics of programs is defined as a transition system on states. Let $\mathcal{M}$ be a set of states and $s, t, s' \ldots$ range over $\mathcal{M}$. We assume that we have a projector pc on states that returns the program point associated to the state. For brevity, we write s_i to indicate that s is a state such that $\mathsf{pc}(s) = i$. The operational semantics of programs is given

by a small-step relation $\rightsquigarrow$ between states; $\rightsquigarrow^\star$ is defined as its reflexive and transitive closure. We assume that $\rightsquigarrow$ is suitably related to $\mapsto$, that is if $s_i \rightsquigarrow t_j$ then $i \mapsto j$.

Without loss of generality w.r.t. the goal of this paper, we only consider type-safe programming languages, and programs that are well-typed w.r.t. safety [29], and thus can never be in an incorrect state. More formally, we assume given a type system $\vdash_{\mathrm{safe}}$ which ensures that programs are type safe. In addition, we must reason about safe states, which intuitively are states that are compatible with the type of the program under execution; therefore, we assume given for each typable program a predicate safe_P on states.

Hypothesis 3 (Progress and preservation of safety)**.** *If* $\vdash_{\mathrm{safe}} P$*, i.e.* P *is typable w.r.t.* $\vdash_{\mathrm{safe}}$*, and* $\mathrm{safe}_P(s_i)$*, then* $i \in \mathcal{P}_{\mathrm{exit}}$ *or there is* s' *such that* $s_i \rightsquigarrow s'$ *and* $\mathrm{safe}_P(s')$*.*

From now on, we assume that programs are type-safe.

Policy setting In expressing non-interference policies, it is common to model confidentiality clearances by a security lattice $(\mathcal{S}, \leq)$ whose elements represent the distinct confidentiality levels. Then, the expected security behaviour of a program is captured by a single global policy Γ that assigns confidentiality levels to variables.

For expressing declassification policies, we take a simple generalisation of the non-interference setting: instead of a single policy Γ for the program, we assume that there is a policy $\Gamma[i]$ for each program point in the program. (We postpone to Section 5.2 the correctness conditions that should be satisfied by the family $\Gamma[i]$). The use of local policies for the specification of declassification permits more precision on what is declassified within a code fragment. Different memory policies for each program point in the program allows us to specify when the security level of a variable x is downgraded from its original policy, as given by the security level $\Gamma[\mathsf{entry}](x)$ for x in the initial program point entry.

Definition of delimited non-disclosure The definition of non-interference for sequential languages may be formulated in terms of a relation between inputs and outputs of the program, because non-interference only considers global policies. (in some settings such as abstract non-interference [17], there is no requirement that the initial and final policies coincide; nevertheless, these definitions do not aim at enforcing local policies).

When defining localised declassification policies, even in sequential settings, the security definition cannot be given in terms of inputs and outputs. This is because memory policies are local and should be respected not only in input/output states but also in intermediate states. Furthermore, we need to define a behavioural equivalence that does not necessarily correspond to any program trace, since we "reset" the memory in some intermediate states [12]. We begin by defining a notion of bisimulation that will characterise the notion of security; in order to reflect the local nature of policies, bisimulation is defined w.r.t. an indexed family $(\sim_{\Gamma[i]})_{i \in \mathcal{P}}$ of symmetric and transitive relations on states.

Definition 1 (DND Bisimulation)**.** *A DND bisimulation is a symmetric relation* $\mathcal{R}$ *between program points in* $\mathcal{P}$ *such that for every* $i, j \in \mathcal{P}$*, if* $i \,\mathcal{R}\, j$ *then for all* s_i*,* t_j *and* $s'_{i'}$ *s.t.*

$$s_i \rightsquigarrow s'_{i'} \wedge s_i \sim_{\Gamma[i]} t_j \wedge \mathrm{safe}_P(t_j)$$

there exists $t'_{j'}$ *such that:*

$$t_j \rightsquigarrow^* t'_{j'} \wedge s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'} \wedge i' \,\mathcal{R}\, j'$$

This notion of bisimulation follows the work of non-disclosure [2]: two program points i and j are related if starting with memories that are equal according to the local memory policy, a transition of the first memory $s_i \rightsquigarrow s'_{i'}$ is matched by zero or more transitions of t_j and the program points of the final memories are bisimilar, and the final memories are related by the global memory policy $\Gamma[\mathsf{entry}]$.

Let $\approx$ be the largest DND bisimulation.

Definition 2 (Delimited non-disclosure)**.** *A program* P *satisfies the delimited non-disclosure policy if* $\mathsf{entry} \approx \mathsf{entry}$*.*

The definition of delimited non-disclosure is termination-sensitive, in contrast to other declassification policies such as localised delimited release [4]. As it is usual in language-based security, the question of whether or not to use the termination-sensitive or termination-insensitive version of the security notion depends on the adversary model. Nevertheless, one could get a termination-insensitive version of DND by adding as hypothesis to the bisimulation that executions starting in s_i and t_j terminate.

We conclude this section with a brief remark about the nature of delimited non-disclosure. As announced, the policy is intended to capture declassification along the *what* and *where* dimensions. While the use of local policies clearly indicates that delimited non-disclosure supports the *where* aspect of declassification, it is not immediate from the definition to which extent the *what* dimension is supported. Clearly, local policies offer the opportunity to declassify variables, but do not explicitly mention the possibility of declassifying expressions. However, we are targeting unstructured languages in which intermediate computations are stored in intermediate memories, typically variables, and therefore it is sufficient to declassify variables instead of expressions. Furthermore, it is always possible to rewrite programs in a semantics-preserving fashion so that declassification of an expression e can be reduced to

declassification on a freshly introduced variable that stores the result of e. This is further elaborated in the next section, where we provide an example of this transformation.

4. Examples

In this section we introduce a simple sequential language to illustrate our policy and compare it with other declassification policies. To ease comparison with previous works where the local memory is inferred from the program syntax (see e.g. [12, 2, 4]), we consider a structured language, and include a syntactic construct for declassification.

Security lattice For the sake of simplicity, we work with a $L \leq H$ security lattice. Besides, we use h (resp. l) as a program variable whose initial memory policy is H (resp. L).

Language The syntax of the language is defined by the grammar in Figure 1 where n ranges over numbers $N = \{0, 1, \ldots\}$, x ranges over program variables, *op* ranges over arithmetic, boolean, and relational operators. As stated above, we include an explicit command for declassification, namely `declassify` (e) `in` $\{\ c\ \}$, to specify local policies by means of the program syntax.

In order to identify program points, some commands of the language are labelled with natural numbers i. We set $\mathsf{entry} = 1$. Local policies in our language might be directly specified by functions that map program points to memory policies. However, one can also choose to infer local memory policies from the program syntax, as explained in Example 1.

The language semantics is standard, and ommitted. The only non-standard command is `declassify` (e) `in` $\{\ c\ \}$—contrary to the introduction, we do not indicate the security level to which an expression is declassified, since there are only two levels. Informally, this command does not affect the semantics (its semantics is equivalent to a `skip`).

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid e \; op \; e \\
c & ::= & [\texttt{skip}]^i \mid c \,;\, c \\
 & & \mid [x := e]^i \\
 & & \mid [\texttt{if}\ (\ e\ )\ \texttt{then}\ \{\ c\ \}\ \texttt{else}\ \{\ c\ \}\ ]^i \\
 & & \mid [\texttt{while}\ (\ e\ )\ \texttt{do}\ \{\ c\ \}]^i \\
 & & \mid \texttt{declassify}\ (e)\ \texttt{in}\ \{\ c\ \}
\end{array}
$$

Figure 1. Expression and command syntax

The command `declassify` (e) `in` $\{\ c\ \}$ is used to specify a local policy where the security level of expression e is L in the scope of command c.

In order to capture the *what* dimension of declassification accurately, the command `declassify` (e) `in` $\{\ c\ \}$ declassifies whole expressions instead of single variables. DND can cope with this form of declassification using a simple program transformation, as is explained in Example 2.

Examples Our first example illustrates how local policies may be inferred from the syntax, and from the initial policy.

Example 1 (Local memory policy inference)**.** *Consider the program:*

$$[l_1 := 0]^1 \,;\, \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l_2 := h]^2\ \} \,;\, [l_3 := l_2]^3$$

For such a program, we generate local memory policies as follows:

$$
\begin{array}{l}
\Gamma[1](l_1) = \Gamma[1](l_2) = \Gamma[1](l_3) = L \\
\Gamma[1](h) = H \\
\Gamma[2](l_1) = \Gamma[2](l_2) = \Gamma[2](l_3) = L \\
\Gamma[2](h) = L \\
\Gamma[3] = \Gamma[1]
\end{array}
$$

Program point 3 *is not in the scope of the declassification command therefore its memory policy is the same of program point* 1.

This program complies with the DND policy for Γ *defined above because the direct flow from* h *to* l_2 *produced by assignment at* 2 *is authorised by the local memory policy* $\Gamma[2]$.

Our second example illustrates how delimited non-disclosure could capture declassification of expressions by an appropriate program transformation.

Example 2 (Declassification of expressions in DND)**.** *Expression level declassification can be accomplished in DND by assigning the declassified expression to a fresh variable and replacing expression occurrences by the new variable. For example, if we are interested in declassifying the expression* $h > 0$ *in the program:*

$$\texttt{declassify}\ (h > 0)\ \texttt{in}\ \{\ [\texttt{if}\ (\ h > 0\ )\ \texttt{then}\ \{\ [l := 0]^2\ \}]^1\ \}$$

we transform the program into:

$$
\begin{array}{l}
[h' := h > 0]^1 \,; \\
\texttt{declassify}\ (h')\ \texttt{in}\ \{\ [\texttt{if}\ (\ h'\ )\ \texttt{then}\ \{\ [l := 0]^3\ \}]^2\ \}
\end{array}
$$

where h' *is a fresh variable name, and generate the following memory policies:*

$$
\begin{array}{l}
\Gamma[1](l) = L \\
\Gamma[1](h) = \Gamma[1](h') = H \\
\Gamma[2](l) = \Gamma[2](h') = L \\
\Gamma[2](h) = H \\
\Gamma[3] = \Gamma[2]
\end{array}
$$

The next examples compare delimited non-disclosure with delimited release and its localised version.

Example 3 (Complies with DR and not DND)**.** *Consider the program:*

$$[l := h]^1 \ ; \ \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^2\ \}$$

The program does not comply with DND because command at program point 1 *contains an explicit flow for the initial local memory policy that says that* $\Gamma[1](l) = L$ *and* $\Gamma[1](h) = H$*. This program complies with* delimited release *[34], that only imposes restrictions on* what *is declassified without considering* where *declassification occurs. As discussed in [4] (p.1), a policy based only on the* what *dimension does not qualify for a declassification policy because it already assumes that secrets are known from the program's start. This program is not accepted by LDR.*

Example 4 (Complies with LDR and DND)**.** *Consider the program:*

$[h_2 := 0]^1$;
$[\texttt{if}\ (\ h_1\ )\ \texttt{then}\ \{\ \texttt{declassify}\ (h_1)\ \texttt{in}\ \{\ [l := h_1]^3\ \}\ \}\ \texttt{else}\ \{$
$\quad \texttt{declassify}\ (h_2)\ \texttt{in}\ \{\ [l := h_2]^4\ \}\ \}\ ]^2$

After the execution of this program the final value of l *is the value of* h_1 *even if* h_1 *has not been declassified. Delimited release accepts this program but DND and* localized delimited release *reject it.*

In contrast to delimited non-disclosure, localised delimited release rejects programs that may lead to laundering attacks [4] by allowing variables to be modified before their declassification. Indeed, our policy does not provide by itself any protection against laundering attacks.

We have essentially two reasons for not excluding laundering attacks by definition of DND. First of all, we believe that the indistinguishability of memories property (exclusively expressed by e.g. DND) and the lack of laundering attacks safety property are independent concepts. Hence there is no need to put both concepts together in a single property, as is the case in LDR. Furthermore, this well marked independence between the "indistinguishability" part of LDR and the laundering attack part of LDR leads as to conjecture that, given corresponding local memory policies, programs that comply with termination-sensitive version of LDR also comply with DND and programs that comply with DND and do not have laundering attacks, comply with LDR.

The second and most important reason is that by separating concepts of laundering attacks and DND, it is possible to modularly extend (as shown in Section 5) type systems for non-interference and even more important, construct a proof of soundness of this extension that can be adapted to a series of different languages, constructs, and non-interference type systems. Even more, laundering attacks can be avoided

$$\frac{}{\vdash_{LA} [\texttt{skip}]^i : \emptyset, \emptyset}$$

$$\vdash_{LA} [x := e]^i : \{x\}, \emptyset$$

$$\frac{\vdash_{LA} C_1 : U_1, V_1 \qquad \vdash_{LA} C_2 : U_2, V_2 \qquad U_1 \cap V_2 = \emptyset}{\vdash_{LA} C_1 \ ; \ C_2 : U_1 \cup U_2, V_1 \cup V_2}$$

$$\frac{\vdash_{LA} C_1 : U, V \qquad \vdash_{LA} C_2 : U, V}{\vdash_{LA} [\texttt{if}\ (\ e\ )\ \texttt{then}\ \{\ C_1\ \}\ \texttt{else}\ \{\ C_2\ \}\ ]^i : U, V}$$

$$\frac{\vdash_{LA} C : U, V \quad U \cap V = \emptyset}{\vdash_{LA} [\texttt{while}\ (\ e\ )\ \texttt{do}\ \{\ C\ \}]^i : U, V}$$

$$\frac{\vdash_{LA} C : U, V}{\vdash_{LA} \texttt{declassify}\ (x)\ \texttt{in}\ \{\ C\ \} : U, V \cup \{x\}}$$

$$\frac{\vdash_{LA} C : U, V \qquad U \subseteq U' \qquad V \subseteq V'}{\vdash_{LA} C : U', V'}$$

Figure 2. Effect system against laundering

by a simple effect system which ensures that variables that are declassified were not previously updated. To show this, we define a type system in Fig. 2 as an example of a type system to prevent laundering attacks. The judgements are of the form $\vdash_{LA} C : U, V$ where C is a command, U is a set of variables which meaning is variables that have been assigned by previous commands but not yet declassified and V is a set of variables which meaning is variables that have been declassified. The typing rules for composite commands, and in particular for loops, put disjointness constraints on the set of previously assigned variables and the set of declassified variables.

Example 5 (Laundering attacks)**.**

$$[h := 0]^1 \ ; \ \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^2\ \}$$

This program is rejected by localised delimited release and accepted by DND. However, the program is rejected by the laundering-attacks type system given in Fig. 2, since the only V *sets typing* $[h := 0]^1$ *must contain* h *by the rule of assingments, and the only* U *sets typing the declassify construct must contain* h *by the rule of declassification constructs. By the rule of sequential composition the* V *set of the assigment and the* U *set of the declassification instruction must not have common elements.*

Our next example suggests that it may be possible to encode localised delimited release using DND. We conjecture that a terminating program is accepted by localized delimited release iff its transformation along the process de-

scribed above is accepted by delimited non-disclosure and by the effect system against laundering.

Example 6 (Program complies with LDR but not with DND)**.** Localised delimited release *accepts programs like:*

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^1\ ;\ [l_2 := h]^2\ \}\ ;\ [l_3 := h]^3$$

because it considers that assignment to l_3 is "intuitively" secure because the program can be safely transformed into:

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^1\ ;\ [l_2 := h]^2\ \}\ ;\ [l_3 := l_2]^3$$

For DND, assignments 1 and 2 are valid because they are made in the scope of the declassification of variable h, but assignment 3 does not comply with our policy.

Notice that, if necessary, the original program can be transformed to be accepted by DND by extending the declassification region until the end of the program:

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^1\ ;\ [l_2 := h]^2\ ;\ [l_3 := h]^3\ \}$$

We conclude this section by a comparison with non-disclosure, which is closely related to DND. The main difference is that DND permits a fine-grained relaxation of non-interference by authorising exceptional flows not from *all* high variables but from user defined sets of *high* variables. In contrast, non-disclosure is based on a global security lattice G that dynamically evolves to interpret locally induced flow policies. These policies introduce new flow relations in G in the context of sets of program points. Outside these sets, the global security lattice remains as it was at the beginning of the program.

The following example compares non-disclosure [2] and delimited non-disclosure, and suggests a possible encoding of ND in terms of DND. In contrast to non-disclosure, we assume that the security lattice does not change at different points of the program but variable confidentiality levels change. Thus, our security policy is not expressed by means of flows between principals (that determine the security lattice) but rather by local memory policies.

Example 7 (Non-disclosure)**.** *The flow declaration construct of non-disclosure*

$$\texttt{flow}\ (\mathtt{p}_2 \prec \mathtt{p}_1)\ \texttt{in}\ c$$

is introduced to express local lattice modifications. Here c is a statement into which the original security lattice induced by principals $\{p_2, p_1\}$ is modified to permit flows from p_2 to p_1. Let's name elements in the original lattice as

$$H = \{\}$$
$$H_1 = \{p_1\}$$
$$H_2 = \{p_2\}$$
$$L = \{p_1, p_2\}$$

where $\leq$ is given by the subset relation. The new lattice induced by the flow construct above, treats H_2 as L (security level H_2 dissapears from the lattice).

In our policy the security lattice is not modified, instead the initial memory policy $\Gamma[1]$ is adapted at each program point in order to reflect local relaxations of non-interference.

Therefore, the effect of the above flow construct, in terms of our policy, is that for all variable x such that $\Gamma[1](x) = H_2$ we have $\Gamma[j](x) = L$ if j is a program point for statement c.

For example, if the program has variables h and h' with $\Gamma[1](h) = \Gamma[1](h') = H_2$, then $\texttt{flow}\ (\mathtt{p}_2 \prec \mathtt{p}_1)\ \texttt{in}\ c$ can be expressed in our language as:

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ \texttt{declassify}\ (h')\ \texttt{in}\ \{\ c\ \}\ \}$$

The flow construct of non-disclosure is expressed in our setting as a declassification of all variables belonging to the security levels induced by the new flow.

5. Enforcing delimited non-disclosure

The purpose of this section is to provide a modular definition and soundness proof of an information flow type system that enforces delimited non-disclosure. For simplicity, we fix the lattice of security levels to $\mathcal{S} = \{L, H\}$ with $L \leq H$ and assume that security policies are functions that attach security levels to program variables. (However the main result applies to arbitrary lattices, the simplification to two level lattices applies to the hypotheses on NI, and these hypotheses can be generalized as shown in e.g. [6].)

The order on security levels can be extended pointwise to security policies; by abuse of notation, we let $\leq$ denote the order between policies.

5.1 Assumptions on NI typing

Our starting point is a type system $\vdash_{NI}$ designed for enforcing non-interference. The type system operates on a program P annotated with control dependence regions $(\mathsf{region}, \mathsf{junction})$, and is parameterised by a security environment se that maps program points to levels, a policy Γ that maps variables to security levels, and a type S; the exact nature of types does not need to be specified. For readability, typing judgements are written in the following form:

$$\Gamma, S \vdash_{NI} i$$

where Γ is a policy, S is a type, i is a program point. All other parameters are left implicit.

The principal hypotheses that we make on $\vdash_{NI}$ take the form of unwinding statements. The first unwinding hypothesis states that execution locally respects state equivalence, i.e. if we start from two equivalent states that point to the same instruction and perform one step of execution, then the two resulting states are also equivalent. Additionally, it

makes some technical assumption about the program counters of the resulting states: either they coincide, or the initial states were pointing to a high branching instruction.

In order to formulate the notion of high branching instruction, we say that k has a high region, written $\mathsf{highregion}(k)$, iff k is a branching point, and $se(j) = H$ for every $j \in \mathsf{region}(k)$. Moreover, we write $i \in \mathsf{highregion}(k)$ to mean $\mathsf{highregion}(k)$ and $i \in \mathsf{region}(k)$.

Hypothesis 4 (LowNI)**.** *Assume that* $\Gamma, S \vdash_{NI} i$*, and* $s_i \sim_\Gamma t_i$*, and* $s_i \rightsquigarrow s'_{i'}$ *and* $t_i \rightsquigarrow t'_{j'}$*. Then* $s'_{i'} \sim_\Gamma t'_{j'}$ *and one of the following holds:*

- $i' = j'$*, or*
- $\mathsf{highregion}(i)$*, and* $i', j' \in \mathsf{region}(i) \cup \{\mathsf{junction}(i)\}$.

The second hypothesis states that executing an instruction in a high control dependence region does not modify the observable part of a state. It corresponds to the step preserving unwinding property of [30].

Hypothesis 5 (HighNI)**.** *Assume that* $\mathsf{highregion}(k)$ *and let* $i, i' \in \mathsf{region}(k)$*. Assume that* $s_i \rightsquigarrow s'_{i'}$ *and* $\Gamma, S \vdash_{NI} i$*. Then* $s_i \sim_\Gamma s'_{i'}$.

Using the above lemmas, one can prove that typable programs are non-interfering.

Definition 3. *A program P is typable with type S w.r.t. Γ, written $\Gamma, S \vdash_{NI} P$, iff for every program point i, we have $\Gamma, S \vdash_{NI} i$.*

Using an adaptation of the general scheme of [7], one can prove that, under the hypotheses of this section, typable programs are non-interfering. More precisely, if P is typable w.r.t. Γ, then P is non interferent w.r.t. Γ, i.e. for all $s_{\mathsf{entry}}, t_{\mathsf{entry}}$:

$$\left.\begin{array}{l} s_{\mathsf{entry}} \sim_\Gamma t_{\mathsf{entry}} \\ s_{\mathsf{entry}} \rightsquigarrow^\star s'_i \\ t_{\mathsf{entry}} \rightsquigarrow^\star t'_j \\ i, j \in \mathcal{P}_{\mathrm{exit}} \end{array}\right\} \Rightarrow s'_i \sim_\Gamma t'_j$$

5.2 Typing delimited non-disclosure

The goal of this section is to formulate the DND-type system and to prove that, under some mild hypotheses, programs that are typable by the DND-type system verify the delimited non-disclosure policy. We begin by defining the type system.

The type system for delimited non-disclosure is very similar to the type system for non-interference, but is parameterised by a family of policies $(\Gamma[i])_{i\in\mathcal{P}}$. Like the type system for non-interference, the type system for delimited non-disclosure is parameterised by control dependence regions $(\mathsf{region}, \mathsf{junction})$, a security environment se and a type S.

Definition 4. *Let $(\Gamma[i])_{i\in\mathcal{P}}$ be an indexed set of local policies. A program P is typable with type S w.r.t. $(\Gamma[i])_{i\in\mathcal{P}}$, written $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$, iff for every program point i, we have $\Gamma[i], S \vdash_{NI} i$.*

The DND type system is conservative: intuitively, programs without declassification (i.e. without different local policies) that are typable by the DND type system are non-interferent programs. However, $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$ does not necessarily imply that $\Gamma[i], S \vdash_{NI} P$ for some $i \in \mathcal{P}(P)$.

Note that, at such an abstract level, there is a strong similarity between our type system for delimited non-disclosure, and flow-sensitive type systems [20], since they both rely on a family of policies $(\Gamma[i])_{i\in\mathcal{P}}$. However, the type system for delimited non-disclosure makes different assumptions on this family: see Hypothesis 7.

Termination Delimited non-disclosure is termination sensitive and the type system must reject any program that has loops whose termination behaviour is influenced by confidential data. Therefore, we must of a program analysis loop that detects that the branching point i is a loop, and we assume that the type system guarantees that all high loops terminate. In the sequel, we write $\mathsf{loop}(i)$ if the analysis detects that i is a loop, see e.g. [28] for a definition of such an analysis.

Hypothesis 6 (Termination of while loops)**.** *Assume $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$, and let i be a program point such that* $\mathsf{highregion}(i)$ *and* $\mathsf{loop}(i)$*. Then* $\mathsf{junction}(i)$ *is defined. Besides, if* $j \in \mathsf{region}(i)$*, and s_j is safe, i.e. $\vdash_{\mathrm{safe}} s_j$, then there exists $t_{j'}$ such that $s_j \rightsquigarrow^\star t_{j'}$, and $j' = \mathsf{junction}(i)$.*

The hypothesis states that high loops terminate normally (in contrast to abrupt termination caused by a return in a loop). This condition can be brutally enforced by typing rules that reject all high loops [36, 2]. However, Gérard Boudol [11] observed that such typing rules can be largely improved by being parameterised by a termination analysis (see e.g. [15] for an advanced analysis of this kind).

Note that one can strengthen the hypothesis by not requiring that i is a loop, using Hypothesis 2.

Lemma 1 (Exit from high guards)**.** *Assume $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$, and let i be a program point such that* $\mathsf{highregion}(i)$ *and* $\mathsf{junction}(i)$ *is defined. If* $j \in \mathsf{region}(i)$*, and s_j is safe, i.e. $\vdash_{\mathrm{safe}} s_j$, then there exists $t_{j'}$ such that $s_j \rightsquigarrow^\star t_{j'}$, and $j' = \mathsf{junction}(i)$.*

Proof. If $\mathsf{loop}(i)$, then we apply directly Hypothesis 6. Otherwise, one can apply Hypotheses 1 and 2, together with progress (Hypothesis 3). □

We use Lemma 1 in the proof of Theorem 1 below.

Correct memory policies In order to be able to prove soundness of the DND type system, we impose mild restrictions on families of local memory policies $(\Gamma[i])_{i\in\mathcal{P}}$. These restrictions can be verified automatically independently of the DND type system.

Hypothesis 7 (Correct memory policies)**.** *A family of memory policies $(\Gamma[i])_{i\in\mathcal{P}}$ is correct if for all $i, j \in \mathcal{P}$:*

1. *for every variable x, we have $\Gamma[i](x) \leq \Gamma[\mathsf{entry}](x)$;*
2. *if $\mathsf{highregion}(i)$ and $j \in \mathsf{region}(i) \cup \{\mathsf{junction}(i)\}$, then $\Gamma[j] = \Gamma[i]$.*

The first item says that a memory policy for a variable x can be downgraded (declassified) but not upgraded. The second item says that inside high control dependence region and up to the junction point, local policies remain unchanged; this assumption is in line with previous work on disallowing declassification inside high branching statements, and rightfully rejects programs such as

$$[\texttt{if}\ (\ h\ )\ \texttt{then}\ \{\ \texttt{declassify}\ (h_1)\ \texttt{in}\ \{\ [l := h_1]^2\ \}\ \}]^1$$

which leaks the value of h through the knowledge of whether h_1 has been declassified or not (see Example 4).

Hypothesis 8 (Monotonicity of $\sim$)**.** *If $\Gamma[i] \leq \Gamma[j]$ and $s \sim_{\Gamma[i]} t$ then $s \sim_{\Gamma[j]} t$.*

The hypothesis guarantees monotonicity of $\sim_{\Gamma[i]}$ w.r.t. the order of local memory policies.

Soundness proof To prove that the DND type system enforces delimited non-disclosure, we exhibit a DND bisimulation $\mathcal{B}$ that satisfies $\mathsf{entry}\ \mathcal{B}\ \mathsf{entry}$. The relation $\mathcal{B}$ is defined inductively by the clauses

$$\frac{}{i\ \mathcal{B}\ i} \qquad \frac{j\ \mathcal{B}\ i}{i\ \mathcal{B}\ j}$$

$$\frac{i, j \in \mathsf{highregion}(k)}{i\ \mathcal{B}\ j}$$

$$\frac{i \in \mathsf{highregion}(k) \quad j = \mathsf{junction}(k)}{i\ \mathcal{B}\ j}$$

Since $\mathcal{B}$ is reflexive, we obviously have $\mathsf{entry}\ \mathcal{B}\ \mathsf{entry}$.

In the sequel, we assume that all aforementioned hypotheses are satisfied.

Theorem 1 (Soundness of $\vdash_{DND}$)**.** *If $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$, then P complies with the delimited non-disclosure policy w.r.t. $(\Gamma[i])_{i\in\mathcal{P}}$.*

Proof. We show that $\mathcal{B}$ is a DND bisimulation. Assume that $i\mathcal{B}j$. There are four cases to treat:

- If $i = j$. Let s_i and t_i be states s.t. $s_i \rightsquigarrow s'_{i'}$ and $s_i \sim_{\Gamma[i]} t_i$. Suppose that $\mathrm{safe}_P(t_j)$. By progress (Hypothesis 3), either $i \in \mathcal{P}_{\mathrm{exit}}$ or there exists $t'_{j'}$ such that $t_i \rightsquigarrow t'_{j'}$. Since $s_i \rightsquigarrow s'_{i'}$, we have $i \notin \mathcal{P}_{\mathrm{exit}}$, and thus there exists $t'_{i'}$ such that $t_i \rightsquigarrow t'_{j'}$. By locally respects unwinding (Hypothesis 4), $s'_{i'} \sim_{\Gamma[i]} t'_{j'}$ and $i' = j'$, or $i', j' \in \mathsf{region}(i) \cup \{\mathsf{junction}(i)\}$. In all cases, we have $i'\mathcal{B}j'$.

  Furthermore, by Hypothesis 7 $\Gamma[i] \leq \Gamma[\mathsf{entry}]$ and by Hypothesis 8 $s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'}$, so we are done.

- If $i, j \in \mathsf{highregion}(k)$ for some k. Let $s_i \sim_{\Gamma[i]} t_j$, and assume that $s_i \rightsquigarrow s'_{i'}$. By step preserving unwinding (Hypothesis 5), $s'_{i'} \sim_{\Gamma[i]} t_j$. By monotonicity of local policies (Hypothesis 7), $s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_j$. Furthermore, exit through junction (Hypothesis 1) ensures that $\mathsf{junction}(i)$ is the unique exit point of $\mathsf{region}(i)$, therefore either $i' \in \mathsf{region}(k)$ or $i' = \mathsf{junction}(k)$. In both cases, $i'\mathcal{B}j$.

- If $i \in \mathsf{highregion}(k)$ and $j = \mathsf{junction}(k)$ for some k. This case is similar to the above (except for the fact that if $i' = \mathsf{junction}(k)$ we use the reflexivity of $\mathcal{B}$ to conclude that $i'\mathcal{B}j$).

- If $j \in \mathsf{highregion}(k)$ and $i = \mathsf{junction}(k)$ for some k. Let $s_i \sim_{\Gamma[i]} t_j$, and assume that $s_i \rightsquigarrow s'_{i'}$. By progress (Hypothesis 3) and exit from high guards (Lemma 1), and by exit through junction (Hypothesis 1), there exists a sequence

  $$t_j \rightsquigarrow u^1_{k_1} \rightsquigarrow \ \ldots\ \rightsquigarrow u^l_{k_l} \rightsquigarrow u'_i$$

  such that $k_1 \ldots k_l \in \mathsf{region}(i)$. By repeatedly applying the step preserves unwinding (Hypothesis 5), appealing to the correctness of memory policy (Hypothesis 7), which ensures that policy do not vary in high regions, and the transitivity of state equivalence, we conclude that $t_j \sim_{\Gamma[k]} u'_i$. Since $\Gamma[k] = \Gamma[i]$ by correctness of memory policy (Hypothesis 7), we have by transitivity $s_i \sim_{\Gamma[i]} u'_i$, and can conclude as in the first case.

□

6. Case study: Java Virtual Machine

The objective of this section is to apply our results to a minimal fragment of the JVM. We also establish type-preserving compilation w.r.t. a type system for the language of Section 4. Finally, we discuss the applicability of the method to a larger fragment of the JVM.

$instr$	::=	binop op	binary operation on stack
	|	push v	push value on top of stack
	|	load x	load value of x on stack
	|	store x	store top of stack in variable x
	|	ifeq j	conditional jump
	|	goto j	unconditional jump

Figure 3. $\mathrm{JVM}_{\mathcal{I}}$ **instructions**

6.1 Language and policy

For brevity, we consider a fragment called $\mathrm{JVM}_{\mathcal{I}}$, whose instruction set is given in Figure 3; we use op to range over binary operations, v over values, x over variables, and j over program points.

The operational semantics of $\mathrm{JVM}_{\mathcal{I}}$ programs is standard, and given by a small-step relation $\rightsquigarrow$ that represents one step execution of the virtual machine. States can either be intermediate, in which case they consist of an operand stack, a memory, and a program counter, or final, in which case they consist of a memory. We use $\langle i, \rho, os\rangle$ to denote an intermediate state with program counter i, memory ρ and operand stack os. Final states are simply identified with memories.

To instantiate delimited non-disclosure to $\mathrm{JVM}_{\mathcal{I}}$ programs, we must first define local policies. A local policy is simply a mapping from variables to levels. We assume given a policy $\Gamma[i]$ for each program point i.

Next, we define an indexed family of partial equivalence relations between states. This involves defining equivalence between memories, and between operand stacks.

Definition 5. *Two memories μ and μ' are equivalent w.r.t. Γ, written $\mu \sim^{\mathsf{Mem}}_{\Gamma} \mu'$, iff $\mu(x) = \mu'(x)$ for every variable x such that $\Gamma(x) = L$.*

Equivalence between operand stacks is defined relative to stack types. (There are both weaker and stronger notions of operand stack equivalence; see [7] for a discussion on these notions).

Definition 6. *The relation $os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2$, where $st_1, st_2 \in \mathcal{S}^*$, is defined inductively, together with the inductively defined auxiliary relation $\mathsf{high}(os, st)$, in Figure 4.*

Finally, state equivalence is defined in the obvious way.

Definition 7. *Let $S : \mathcal{P} \to \mathcal{S}^*$. Two states $s = \langle i, \rho, os\rangle$ and $s' = \langle i', \rho', os'\rangle$ are equivalent, written $s \sim^{\mathsf{State}} s'$, iff $\Gamma(i) = \Gamma(i')$ and $\rho \sim^{\mathsf{Mem}}_{\Gamma(i)} \rho'$ and $os \sim^{\mathsf{Stk}}_{S(i),S(i')} os'$.*

To conclude with the definition of delimited non-disclosure, one needs to define the notion of safe state. In our setting, a safety type assigns to each program point a natural number that represents the height of its operand stack, and a state is safe if its operand stack has the correct height w.r.t. its program counter. The safety type system tracks the height of the operand stack, and ensures that jumps are correct, i.e. remain within the program code. It is easy to show Hypothesis 3, i.e. that safe states enjoy progress. In a more general setting, one can define safe states using the work of Freund and Mitchell [16], who formalized a safety type system for the JVM, and showed that safe programs enjoy progress.

$$\frac{\mathsf{high}(os_1, st_1) \qquad \mathsf{high}(os_2, st_2)}{os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2}$$

$$\frac{os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2}{v :: os_1 \sim^{\mathsf{Stk}}_{L::st_1,L::st_2} v :: os_2}$$

$$\frac{os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2}{v_1 :: os_1 \sim^{\mathsf{Stk}}_{H::st_1,H::st_2} v_2 :: os_2}$$

$$\frac{}{\mathsf{high}(\epsilon, \epsilon)} \qquad \frac{\mathsf{high}(os, st)}{\mathsf{high}(v :: os, H :: st)}$$

Figure 4. Operand stack equivalence

6.2 Type system

The type system is expressed by rules of the form

$$i \overset{JVM}{\vdash} st \Rightarrow st'$$

where i is a program point and $st, st' \in \mathcal{S}^\star$ are stacks types. The rules are given in Figure 5, and assume that programs come equipped with control dependence regions (cdr), and a security environment. The rules exactly match the rules of [6], except that:

- the rules for load and store use the local policy;
- the rule for ifeq rejects high loops, and is instantiated to the case where the stack is empty after execution (which is the case for compiled programs, see [23]).

The typing rules of $\mathrm{JVM}_{\mathcal{I}}$ can be viewed as an instance of the generic type system. Indeed, define a type to be a map S from program points to stack types. Then, we define $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$ iff the following holds:

- $S(\mathsf{entry})$ is the empty stack;
- for every i s.t. $se(i) = H$, the stack $S(i)$ is high (i.e. all elements of $S(i)$ are equal to H);

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash_{DND} st \Rightarrow se(i) :: st}$$

$$\frac{P[i] = \mathsf{binop}\ op}{i \vdash_{DND} k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2) :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \Gamma_i(x)}{i \vdash_{DND} k :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash_{DND} st \Rightarrow (\Gamma_i(x) \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash_{DND} st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{return} \qquad se(i) = L}{i \vdash_{DND} k :: st \Rightarrow \epsilon}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \mathsf{loop}(i) \Rightarrow k = L \quad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash_{DND} k :: \epsilon \Rightarrow \epsilon}$$

Figure 5. Transfer rules for $\mathrm{JVM}_{\mathcal{I}}$ instructions

- if $i \mapsto j$, then $i \overset{JVM}{\vdash} S(i) \Rightarrow st$ for some $st \leq S(j)$.

Under this definition, and restricting ourselves to constant families of policies (i.e. families of policies $(\Gamma[i])_{i\in\mathcal{P}}$ s.t. $\Gamma[i] = \Gamma[j]$ for all i and j), the notion of typable program w.r.t. $\vdash_{NI}$ coincides with the notion of typable program in [6]. Furthermore, one can use our construction of $\vdash_{DND}$, Theorem 1 and our earlier results in the proof of non-interference for $\mathrm{JVM}_{\mathcal{I}}$ to conclude that $\vdash_{DND}$ enforces delimited non-disclosure.

Theorem 2. *Let $(\Gamma_i)_{i\in\mathcal{P}}$ be correct policies. Let P be a safe program such that $(\Gamma_i)_{i\in\mathcal{P}}, S \vdash_{\mathrm{DND}} P$. If* (region, junction) *satisfy the SOAP properties (given in Figure 6), then P satisfy delimited non-disclosure.*

We briefly indicate why the hypotheses of Section 5 hold. Exit through junction (Hypothesis 1) corresponds exactly to the property **SOAP2**, whereas no return before junction (Hypothesis 2) corresponds exactly to the property **SOAP3**.

Progress and preservation of safety (Hypothesis 3) hold as explained above.

The unwinding statements (Hypotheses 4 and 5) are direct consequences of the unwinding lemmas proved in [6]; note that the unwinding statements are proved using the **SOAP** properties.

Hypothesis 6 holds by definition of the typing rule for ifeq, which prevents highregion(i) and loop(i) from holding simultaneously.

SOAP1 for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in \mathsf{region}(i)$ or $k = \mathsf{junction}(i)$;

SOAP2 for all program points i, j, k, if $j \in \mathsf{region}(i)$ and $j \mapsto k$, then either $k \in \mathsf{region}(i)$ or $k = \mathsf{junction}(i)$;

SOAP3 for all program points i, j, if $j \in \mathsf{region}(i)$ and $j \in \mathcal{P}_{\mathrm{exit}}$ then $\mathsf{junction}(i)$ is undefined.

Figure 6. SOAP properties

Finally, correctness of memories (Hypothesis 7) is an assumption of the theorem, and monotonicity (Hypothesis 8) holds trivially.

6.3 Type-preserving compilation

In this section, we focus on preservation of typability by compilation. The benefits of type preservation are two-fold: they guarantee program developers that their programs written in an information flow aware programming language will be compiled into executable code that will be accepted by a security architecture that integrates an information flow bytecode verifier. Conversely, they guarantee code consumers of the existence of practical tools to develop applications that will provably meet the policy enforced by their information flow aware security architecture.

We consider the source language introduced in Section 4, and define a declassification type system. The type system is parameterized by a family $(\Gamma[i])_i$ of local policies, in this case one policy per label. As already explained in Example 1, these local policies can be inferred from the initial policy, and from the program syntax. (Alternatively, one could formulate a type system that is parameterized by a single policy, that corresponds to the policy at the entry point of the program, and use the type system to track the local changes in the policy.)

Furthermore, the local policies $(\Gamma_i)_{i\in\mathcal{P}}$ for $T(P)$ are generated from the initial policy of P (that is the policy of the entry point of P) and from the `declassify` (.) `in` { . } constructs, as explained in Example 1.

The type system for the source language is given by the rules given in Figs. 7 and 8. Notice that since we have local policies $(\Gamma[i])_i$ for each program point, the rule for declassification corresponds exactly to typability of C and the type system is very similar to a non-interference type system except because it uses a set of local policies instead of a unique global policy.

Notice furthermore that the typing rules are restricted to programs in which only variables are declassified. In order to extend typing to programs that do not meet this restriction, we use the source-to-source trans-

$$\frac{}{(\Gamma[i])_i \vdash_{DND} n : L}$$

$$\frac{}{(\Gamma[i])_i \vdash_{DND} x : \Gamma(x)}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e_1 : k \qquad (\Gamma[i])_i \vdash_{DND} e_2 : k}{(\Gamma[i])_i \vdash_{DND} e_1 \; op \; e_2 : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : k_1 \qquad k_1 \leq k_2}{(\Gamma[i])_i \vdash_{DND} e : k_2}$$

Figure 7. Typing rules for expressions

$$\frac{}{(\Gamma[i])_i \vdash_{DND} [\texttt{skip}]^i : L}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : \Gamma_i(x)}{(\Gamma[i])_i \vdash_{DND} [x := e]^i : \Gamma(x)}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : k \quad (\Gamma[i])_i \vdash_{DND} C_1 : k \qquad (\Gamma[i])_i \vdash_{DND} C_2 : k}{(\Gamma[i])_i \vdash_{DND} [\texttt{if} \; (\; e \;) \; \texttt{then} \; \{ \; C_1 \; \} \; \texttt{else} \; \{ \; C_2 \; \}]^i : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} C : k}{(\Gamma[i])_i \vdash_{DND} \texttt{declassify} \; (x) \; \texttt{in} \; \{ \; C \; \} : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : L \qquad (\Gamma[i])_i \vdash_{DND} C : L}{(\Gamma[i])_i \vdash_{DND} [\texttt{while} \; (\; e \;) \; \texttt{do} \; \{ \; C \; \}]^i : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} C : k \qquad k' \leq k}{(\Gamma[i])_i \vdash_{DND} C : k'}$$

Figure 8. Typing rules for commands

formation introduced in Example 2. This transformation replaces `declassify` (e) `in` $\{ \; c \; \}$ by $x := e;$ `declassify` (x) `in` $\{ \; c' \; \}$, where x is a fresh variable and c' is recursively obtained from applying the same transformation to $c[e/x]$. This transformation is semantics-preserving provided variables in e are not modified in c. In the sequel, we denote by $T(P)$ the result of applying this transformation to P.

We consider a non-optimizing compiler $[\![.]\!]$. Its definition on programs is standard, except for the statement `declassify` (x) `in` $\{ \; c \; \}$, which is compiled to $[\![c]\!]$ (that is, declassify statements are ignored by compilation). The compiler is extended to programs that delassify expressions by composition with the transformation T.

As the bytecode type system uses both a cdr structure $(\mathsf{region}, \mathsf{junction})$, a security environment se, and local policies $(\Gamma[i])_{i \in \mathcal{P}}$, the compiler must also generate this additional information. Furthermore, the generated information must ensure that $[\![P]\!]$ is typable w.r.t. $(\mathsf{region}, \mathsf{junction})$, $(\Gamma[i])_{i \in \mathcal{P}}$ and se. The cdr structure and security environment of the compiled programs can be defined as in earlier works on type-preserving compilation, e.g. [8].

The compiler maps every labeled statement in the source programs to a set of program points. The local policy of these program points is inherited from the local policy of the label of their corresponding source statement.

Theorem 3 (Typability Preservation)**.** *Let P be a source program with correct memory policies. Assume that $T(P)$ is typable by the DND source type system. Then $[\![P]\!]$ is a typable bytecode program (w.r.t. the generated information).*

In addition, the generated cdr structure $(\mathsf{region}, \mathsf{junction})$ satisfies the SOAP properties and the generated local policies $(\Gamma[i])_{i \in \mathcal{P}}$ are correct. Therefore, one can conclude by Theorem 2 that the compiled program verifies delimited non-disclosure w.r.t. the family of local policies generated by the compiler.

Section 4 also presents an effect system to prevent laundering attacks. Although we refrain from doing so here, it is possible to define a similar effect system for the $\mathrm{JVM}_{\mathcal{I}}$ and show that compilation preserves typability wr.t. this system.

7 Discussion

7.1 Objects, exceptions, and methods

Our method has been described and instantiated in a representative, but simplified, setting. Leveraging it to the sequential fragment of the Java Virtual Machine does not pose any major difficulty, but involves a significant amount of technicalities. Fortunately, these technicalities were already handled in the NI work on the JVM.

The first class of technicalities arises from dealing with object-oriented features. Firstly, information flow type systems for the JVM must rely on security signatures with exception effects to support modular verification, and therefore to remain compatible with bytecode verification. Furthermore, signatures and specifications must be compatible with method overriding. Secondly, in presence of objects, state equivalence is formulated in terms of heap equivalence, which must be carefully handled to avoid flows based on non-opaqueness of pointers [19]. Thirdly, exceptions introduce some additional potential sources of indirect flows, and thus must be accounted for in the type system.

In addition, further technicalities are required to achieve an analysis with sufficient precision. Indeed, the presence of exceptions and object-orientation yields a significant blow-up in the control flow graph of the program, and, if no care is taken, may lead to overly conservative type-based analyses. In order to achieve an acceptable degree of usability,

the information flow type system of [6] relies on preliminary analyses that provide a more accurate approximation of the control flow graph of the program. Typically, the preliminary analyses will perform safety analyses such as class analysis, null pointer analysis, exception analysis, and array out-of-bounds analysis. These analyses drastically improve the quality of the approximation of the control flow graph. In particular, one can define a tighter successor relation $\mapsto$ that leads to more precise control dependence regions; in [6], precision is further increased by indexing $\mapsto$ and control dependence regions by a tag (an exception or a special tag for normal execution).

7.2 Multi-threading

As multi-threading is widely used in applications to mobile code, there is a strong interest in developing enforcement mechanisms for multi-threaded programs. There is a wide range of works that consider information flow policies for multi-threaded source programs, see e.g. [10, 25, 32, 33], and it would be interesting to understand how the modular technique of this paper could be applied to this setting.

Building upon earlier work by Russo and Sabelfeld [31], [9] considers a modular method to devise sound enforcement mechanisms for multi-threaded programs. The central idea of these works is to constrain the behavior of the scheduler so that it does not leak information; it is achieved by giving to the scheduler access to the security levels of program points, and by requiring that the choice of the thread to be executed respects appropriate conditions. As in the present paper, the type system for the concurrent language is defined in a modular fashion from the type system for the sequential language, and the soundness of the concurrent type system is derived from unwinding lemmas for the sequential type system.

The kind of extension presented here is orthogonal to the multi-threaded extension shown in [9], and we believe that modular extensions can be combined to augment policy and language expressivity in a single bytecode verifier that enforces DND for a concurrent JVM. Understanding the intuitive guarantees provided by the extension of DND to concurrent languages and formalizing the details of the combination of [9] with the results of this paper is left for future work.

7.3 Formal proofs

Information flow type systems are complex mechanisms whose soundness proofs are particularly involved, especially when considering permissive declassification policies for real programming languages such as the JVM. As such type systems are designed to complement existing type systems for safety and thus lie at the heart of the Trusted Computing Base (TCB), it is therefore fundamental that their implementation is correct, since flaws in the implementation of a type system can be exploited to launch attacks. In our earlier work [6], we have used the proof assistant Coq to formally verify the soundness of an information flow type system that ensures non-interference for a sequential fragment of the Java Virtual Machine. In addition to providing strong guarantees about the correctness of the type system, the formalization serves as a basis for a Foundational Proof Carrying Code architecture. A distinctive feature of our architecture is that the type system is executable inside higher order logic and thus one can use reflection for verifying certificates within Coq, or extraction to obtain an OCaml implementation of a lightweight information flow checker. As compared to Foundational Proof Carrying Code [3], which is deductive in nature, reflective Proof Carrying Code exploits the interplay between deduction and computation to support efficient verification procedures and compact certificates.

As a benefit of the modularity of our approach, we believe that it is possible to achieve, at a moderate cost, a proof of soundness for our DND information flow type system presented, using the formalization reported in [6]. To be more specific, the Coq development is organized in two parts: a generic part, that derives the soundness of the non-interference type system from the unwinding lemmas, and a specific part, that establishes the unwinding lemmas for a particular language, operational semantics, and type system. We are confident that extending the generic part of the formalization to accomodate DND is direct, as in fact proving Theorem 1 in an abstract setting is direct. Nevertheless, we anticipate a fair amount of bookkeeping in the instantiation: even if there is no conceptual difficulty in programming in Coq a bytecode verifier that enforces DND, the specification of the non-interference type system for the JVM is rather large, and extending (even in the modular fashion) its definition to DND—and thus to have a policy per program point instead of a global policy—will be demanding.

8 Conclusion

Tractable enforcement of declassification policies for bytecode languages is an essential step towards a practical use of language-based security in mobile code. In this paper, we have developed a modular method to extend non-interference type systems to sound information flow type systems for delimited non-disclosure, a security policy that combines the *what* and *where* dimensions of declassification, and that is closely related to policies such as delimited release, localized delimited release, and non-disclosure. As a case study, we have instantiated our results to a sequential

fragment $\mathrm{JVM}_{\mathcal{I}}$ of the Java Virtual Machine, yielding the first sound information flow type system to support declassification for an unstructured language. In addition, we have argued that our approach is scalable to exceptions, objects, and methods. As a final contribution, we have shown that our results on type-preserving compilation readily adapt to declassification.

As future work, we intend to spell out the details of extening our results to a richer language with object-oriented features and concurrency, and to provide machine-checked proofs of our results.

Acknoledgements: We thank Ana Almeida Matos, Gérard Boudol, and anonymous reviewers for providing insightful comments on the final version of this paper. This work is partially funded by the EU project MOBIUS and by the ANR project PARSEC.

References

[1] *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society, 2005.

[2] A. Almeida Matos and G. Boudol. On Declassification and the Non-Disclosure Policy. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

[3] A. W. Appel. Foundational Proof-Carrying Code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247, Washington, DC, USA, 2001. IEEE Computer Society.

[4] A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, New York, NY, USA, 2007. ACM.

[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *29th IEEE Symposium on Security and Privacy*, May 2008.

[6] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In Nicola [27], pages 125–140.

[7] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. Technical report, INRIA, 2007. Extended version of [6]. Available from `http://hal.inria.fr/inria-00106182/`.

[8] G. Barthe, T. Rezk, and D. A. Naumann. Deriving an Information Flow Checker and Certifying Compiler for Java. In *27th IEEE Symposium on Security and Provacy*, pages 230–242. IEEE Computer Society, 2006.

[9] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of Multithreaded Programs by Compilation. In J. Biskup and J. Lopez, editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007.

[10] A. Bossi, C. Piazza, and S. Rossi. Compositional information flow security for concurrent programs. *Journal of Computer Security*, 15(3):373–416, 2007.

[11] G. Boudol. On Typing Information Flow. In D. V. Hung and M. Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2005.

[12] N. Broberg and D. Sands. Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In P. Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2006.

[13] S. Chong and A. C. Myers. Language-based information erasure. In aa [1], pages 241–254.

[14] S. Chong and A. C. Myers. End-to-end enforcement of erasure. In *CSFW*. IEEE Computer Society, 2008. To appear.

[15] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.

[16] S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *J. Autom. Reason.*, 30(3-4):271–321, 2003.

[17] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In N. D. Jones and X. Leroy, editors, *POPL*, pages 186–197. ACM, 2004.

[18] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[19] D. Hedin and D. Sands. Noninterference in the Presence of Non-Opaque Pointers. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 217–229, Washington, DC, USA, 2006. IEEE Computer Society.

[20] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, NY, USA, 2006. ACM.

[21] S. Hunt and D. Sands. Just Forget it – The Semantics and Enforcement of Information Erasure. In *Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008*, LNCS. Springer Verlag, 2008.

[22] N. Kobayashi and K. Shirane. Type-Based Information Analysis for Low-Level Languages. In *APLAS*, pages 302–316, 2002.

[23] X. Leroy. Bytecode verification on Java smart cards. *Software: Practice and Experience*, 32(4):319–340, Apr. 2002.

[24] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In Nicola [27], pages 141–156.

[25] H. Mantel and A. Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-threaded Programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[26] H. Mantel and D. Sands. Controlled Declassification Based on Intransitive Noninterference. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2004.

[27] R. D. Nicola, editor. *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007,*

Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[29] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[30] J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical report, SRI, dec 1992.

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

[32] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–273. Springer-Verlag, July 2003.

[33] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 376–394, Madrid, Spain, Sept. 2002. Springer-Verlag.

[34] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *ISSS*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.

[35] A. Sabelfeld and D. Sands. Dimensions and Principles of Declassification. In aa [1], pages 255–269.

[36] D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

[37] S. Zdancewic. Challenges for Information-flow Security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, August 2004.

21st IEEE Computer Security Foundations Symposium

Tractable Enforcement of Declassification Policies

Gilles Barthe
INRIA Sophia Antipolis
Gilles.Barthe@inria.fr

Salvador Cavadini
INRIA Sophia Antipolis
Salvador.Cavadini@sophia.inria.fr

Tamara Rezk
INRIA Sophia Antipolis and MSR-INRIA Joint Centre
Tamara.Rezk@inria.fr

Abstract

Formalizing appropriate information policies that authorize some controlled form of information release, and providing sound analyses for these policies is a necessary step towards practical applications of language-based security.

We propose a modular method to enhance non-interference type systems to support controlled forms of information release that combine the what *and* where *dimensions of declassification. As a case study, we derive from earlier work on non-interference type systems new type systems that soundly enforce declassification policies for sequential fragments of the Java Virtual Machine.*

Our work provides the first modular method to define sound type systems for declassification policies, and the first instance of a sound type system that supports declassification policies for unstructured languages.

1. Introduction

Non-interference [18] is a baseline information flow policy which ensures that publicly available information does not reveal any information about sensitive data during program execution. Non-interference policies are appealing because of the strong security guarantees they provide (zero information leakage); however, practical confidentiality policies cannot be construed to non-interference, as they almost invariably require some constrained release of information.

An important challenge in information flow language-based security is to capture more accurately the kind of confidentiality policies that are needed in practise and to provide tractable and sound enforcement mechanisms for these policies [37]. Declassification policies [35] are weaker forms of information flow policies that permit some constrained information release along four axes regarding *what* specific information is released, *where* within a specific code fragment it is released, *who* releases it, and *when* it is released.

Our contribution is in the line of designing tractable and sound enforcement mechanisms for declassification policies. More concretely, we provide a modular method for achieving sound type systems for declassification from sound type systems for non-interference, and instantiate our method to a sequential fragment of the Java Virtual Machine.

We consider a declassification policy that combines the *what* and *where* dimensions, called *delimited non-disclosure*, that is closely related to the *non-disclosure* policy proposed by Almeida Matos and Boudol [2] and *localised delimited release* proposed by Askarov and Sabelfeld [4]. Informally, delimited non-disclosure is intended to combine two dimensions in a single construct of the form

$$\mathsf{declassify}\ e : \tau\ \mathsf{in}\ c$$

where e is an expression, τ is a security level, and c is a statement.

The choice of the delimited non-disclosure policy is motivated by modularity; our aim is not to propose a new declassification policy but rather to propose a systematic extension of non-interference soundness results in order to achieve declassification soundness for extended non-interference type systems. This modularity is the most distinctive feature of our work. Indeed, existing proofs of soundness proceed by induction over typing derivations, and reproduce a large part of the proof of soundness of the declassification-free fragment of the type system. Instead, we take advantage of the fact that delimited non-disclosure coincides with non-interference for programs without declassification to build a modular proof of soundness. More precisely, we show that if the type system enforces non-interference on the declassification-free fragment of the

978-0-7695-3182-3/08 $25.00 © 2008 IEEE
DOI 10.1109/CSF.2008.11

language, then it enforces delimited non-disclosure. Our method is based on a successor relation between program points, rather than on the syntactic structure of programs, and relies on the idea of control dependence regions, which over-approximate the scope of branching instructions, i.e. instructions that have two or more successors and that can thus yield implicit flows. Here we adopt a local view of policies, which substantially leverages the applicability of information flow type checking and supports more permissive policies.

A second distinctive feature of our approach is its application to low-level languages. In the context of mobile code security, it is essential for code consumers to be able to verify security policies independently and efficiently on the code they receive. Since mobile code applications are typically downloaded in the form of bytecode programs, it is required that verification operates at this level. However, a large body of existing work on language-based security focuses on source languages, and in fact we are not aware of any sound type system that supports declassification policies for unstructured languages. In fact, existing information flow type systems for unstructured languages typically enforce non-interference, see e.g. [6].

Contributions Our main technical contributions are:

- a modular method to define and prove soundness of declassification type systems;
- a case study in which we define and prove soundness of a type system that enforces delimited non-disclosure for a fragment of the Java Virtual Machine;
- a proof of preservation of typing between a type system that enforces delimited non-disclosure in a high-level language and our type system for the JVM.

Organisation The remaining of the paper is organized as follows. Section 2 discusses closely related work. The delimited non-disclosure policy is introduced in Section 3, and Section 4 provides through examples of programs a more detailed comparison with other declassification policies. Section 5 shows how to construct a sound type system from delimited non-disclosure, starting from a type system for non-interference. In Section 6, we instantiate our method to a sequential fragment of the Java Virtual Machine. Section 7 discusses extensions of our work. We conclude in Section 8.

2 Related work

Sabelfeld and Sands [35] analyse main trends on information declassification, and provide a set of principles to be used as "sanity checks" for declassification models. Here we only focus on very closely related work and we refer to Sabelfeld and Sands' survey for a wider overview.

Intransitive non-interference In *intransitive non-interference* policies, flows from high variables to low ones are mediated by a declassifier (or a downgrader), this way non-interference becomes intransitive [30].

In [26], Mantel and Sands introduce an intransitive non-interference based approach concerned to *where* information is declassified.

In this approach, flow ordering is defined by two components: the lattice of security levels, and an "exceptions" relation on security levels expressing special flow cases.

The use of this special relation is deliberately circumscribed to *declassification assignments*, that is, assignments that are allowed to violate the flow ordering (defined by the lattice structure of security levels) as long they respect the "exceptions" relation. This way, information declassification can take place only at specific program points, i.e. at declassification assignments.

Mantel and Sands also present a bisimulation-based definition of their security policy along with a type system to enforce it in a multi-threaded while language.

Localised delimited release Sabelfeld and Myers [34] present *delimited release*, a security policy to declassify information through the special language operator `declassify(e)` where e is globally considered as declassified (*what* dimension).

Later, Askarov and Sabelfeld [4] extended *delimited release* into *localized delimited release*, a security definition combining *what* and *where* dimensions of information declassification. This is made through language semantics instrumentation with the set of declassified expressions and capturing the scope of `declassify(e)` operators.

Non-disclosure Almeida Matos and Boudol [2] proposed the *non-disclosure policy*, a generalisation of non-interference that supports locally induced flow policies through the use of the special construct `flow` $(\mathtt{A} \prec \mathtt{B})$ `in` `S`, where S is a language expression into which flows from principal A to principal B are authorised. Non-disclosure is classified as a declassification policy of the *where* dimension.

Flow locks Broberg and Sands [12] introduce the notion of *flow lock* to specify local information flow policies. In this approach, each variable has attached the set of system principals (or security levels) that can read it. For each principal in this set it is possible to define conditions under which a principal has the right to access to the variable's

value. These circumstances are represented as locks. Local changes of the global policy are specified by means of special program instructions to open and close locks.

Generalisations of non-interference such as non-disclosure and some forms of intransitive non-interference can be represented by flow locks.

Information erasure Hunt and Sands [21] study the semantics of information erasure, a policy aimed at providing guarantees that certain information is not retained after its intended use. Although it is possible to define erasure policies that cover the what and where dimensions of declassification, Hunt and Sands focus on the where dimension. They show that every erasure property can be encoded as (flow-sensitive) non-interference and provide a language construct to express erasure policies in code blocks. They also provide a *global erasure property* and show that it can be enforced by combinning global non-interference and local (command level) non-interference; and define a type system for enforcing the erasure property in a simple while language with inputs and outputs.

The work of Hunt and Sands is inspired by earlier work by Chong and Myers [13], who considered erasure in combination with declassification. Recently, Chong and Myers [14] proposed an information flow type system that enforces non-interference according to a generalization of non-interference introduced by the authors in [13].

Declassification by logic Banerjee et al. [5] present a powerful way to specify security policies including conditions under which declassification is permitted. The specification mechanism is based on *flowtriples*, a combination of program specification and security typing.

The authors also describe an enforcement mechanism integrating security typing (for declassification-free segments of the program), and relational verification and assertion verification (for instructions sequences in flow triples).

$WHERE$**,** $WHAT_1$ **and** $WHAT_2$ In [24], Mantel and Reinhard propose three security conditions for controlling the *where* and *what* dimensions of declassification. The $WHERE$ condition is similar to intransitive non-interference but it satisfies *monotonicity of release* along with the other three declassification principles from [35]. While both $WHAT_1$ and $WHAT_2$ are applicable to concurrent programs, the former is compositional but does not satisfy *monotonicity of release* principle, and the later satisfies this principle but is not compositional.

The authors provide a type system to enforce *where* and *what* dimensions of declassification in such a way that *all* declassified expressions (what) are allowed to flow to low variables at declassification assignments (where).

3. Delimited Non-Disclosure

In this section, we introduce delimited non-disclosure (DND). We formulate our policy in an abstract setting that can be instantiated to different programming languages.

Program setting We let P range over programs of a given programming language. Each program P has an associated set $\mathcal{P}$ of program points that includes a distinguished entry point entry and a set $\mathcal{P}_{\text{exit}}$ of exit points; we let $i, j, i' \ldots$ range over program points. We assume that for a program P there is a successor relation $\mapsto$ between program points. We let $\mapsto^*$ be the reflexive and transitive closure of $\mapsto$, and assume that, whenever $i \in \mathcal{P}_{\text{exit}}$, there is no successor program point such that $i \mapsto j$. We let $\mathcal{P}^\sharp$ denote the set of branching program points, i.e. those program points that have at least two distinct successors.

One primary target of our work are unstructured programming languages, and therefore we rely on control dependence regions to approximate the scope of branching statements; such control dependence regions have been introduced in the context of non-interference in [22], and used in our earlier work [6]. Formally, we assume given two functions:

$$\begin{array}{lcl} \mathsf{region} & : & \mathcal{P}^\sharp \mapsto \wp(\mathcal{P}) \\ \mathsf{junction} & : & \mathcal{P}^\sharp \rightharpoonup \mathcal{P} \end{array}$$

that respectively provide for each branching program point the set of program points, called region, that execute depending on the branching point; and a junction point (if it exists), that denotes the unique exit point from the region. We assume that these functions correctly over-approximate the scope of branching statements, as formulated in the hypothesis below.

Hypothesis 1 (Exit through junction)**.** *If* $j \in \mathsf{region}(i)$ *and* $k \in \mathcal{P}$ *and* $j \mapsto k$*, then either* $k \in \mathsf{region}(i)$ *or* $k = \mathsf{junction}(i)$.

Furthermore, we assume that the junction point of a region is undefined if the region contains an exit point.

Hypothesis 2 (No return before junction)**.** *If* $j \in \mathsf{region}(i) \cup \mathcal{P}_{\text{exit}}$ *then* $\mathsf{junction}(i)$ *is undefined.*

These hypotheses exactly correspond to the safe over approximation properties **SOAP2** and **SOAP3** introduced e.g. in [6] and used in the case study of Section 6.

The semantics of programs is defined as a transition system on states. Let $\mathcal{M}$ be a set of states and $s, t, s' \ldots$ range over $\mathcal{M}$. We assume that we have a projector pc on states that returns the program point associated to the state. For brevity, we write s_i to indicate that s is a state such that $\mathsf{pc}(s) = i$. The operational semantics of programs is given

by a small-step relation $\rightsquigarrow$ between states; $\rightsquigarrow^\star$ is defined as its reflexive and transitive closure. We assume that $\rightsquigarrow$ is suitably related to $\mapsto$, that is if $s_i \rightsquigarrow t_j$ then $i \mapsto j$.

Without loss of generality w.r.t. the goal of this paper, we only consider type-safe programming languages, and programs that are well-typed w.r.t. safety [29], and thus can never be in an incorrect state. More formally, we assume given a type system $\vdash_{\text{safe}}$ which ensures that programs are type safe. In addition, we must reason about safe states, which intuitively are states that are compatible with the type of the program under execution; therefore, we assume given for each typable program a predicate safe_P on states.

Hypothesis 3 (Progress and preservation of safety)**.** *If $\vdash_{\text{safe}} P$, i.e. P is typable w.r.t. $\vdash_{\text{safe}}$, and $\text{safe}_P(s_i)$, then $i \in \mathcal{P}_{\text{exit}}$ or there is s' such that $s_i \rightsquigarrow s'$ and $\text{safe}_P(s')$.*

From now on, we assume that programs are type-safe.

Policy setting In expressing non-interference policies, it is common to model confidentiality clearances by a security lattice $(\mathcal{S}, \leq)$ whose elements represent the distinct confidentiality levels. Then, the expected security behaviour of a program is captured by a single global policy Γ that assigns confidentiality levels to variables.

For expressing declassification policies, we take a simple generalisation of the non-interference setting: instead of a single policy Γ for the program, we assume that there is a policy $\Gamma[i]$ for each program point in the program. (We postpone to Section 5.2 the correctness conditions that should be satisfied by the family $\Gamma[i]$). The use of local policies for the specification of declassification permits more precision on what is declassified within a code fragment. Different memory policies for each program point in the program allows us to specify when the security level of a variable x is downgraded from its original policy, as given by the security level $\Gamma[\mathsf{entry}](x)$ for x in the initial program point entry.

Definition of delimited non-disclosure The definition of non-interference for sequential languages may be formulated in terms of a relation between inputs and outputs of the program, because non-interference only considers global policies. (in some settings such as abstract non-interference [17], there is no requirement that the initial and final policies coincide; nevertheless, these definitions do not aim at enforcing local policies).

When defining localised declassification policies, even in sequential settings, the security definition cannot be given in terms of inputs and outputs. This is because memory policies are local and should be respected not only in input/output states but also in intermediate states. Furthermore, we need to define a behavioural equivalence that does not necessarily correspond to any program trace, since we "reset" the memory in some intermediate states [12]. We begin by defining a notion of bisimulation that will characterise the notion of security; in order to reflect the local nature of policies, bisimulation is defined w.r.t. an indexed family $(\sim_{\Gamma[i]})_{i \in \mathcal{P}}$ of symmetric and transitive relations on states.

Definition 1 (DND Bisimulation)**.** *A DND bisimulation is a symmetric relation $\mathcal{R}$ between program points in $\mathcal{P}$ such that for every $i, j \in \mathcal{P}$, if $i \,\mathcal{R}\, j$ then for all s_i, t_j and $s'_{i'}$ s.t.*

$$s_i \rightsquigarrow s'_{i'} \wedge s_i \sim_{\Gamma[i]} t_j \wedge \text{safe}_P(t_j)$$

there exists $t'_{j'}$ such that:

$$t_j \rightsquigarrow^* t'_{j'} \wedge s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'} \wedge i' \,\mathcal{R}\, j'$$

This notion of bisimulation follows the work of non-disclosure [2]: two program points i and j are related if starting with memories that are equal according to the local memory policy, a transition of the first memory $s_i \rightsquigarrow s'_{i'}$ is matched by zero or more transitions of t_j and the program points of the final memories are bisimilar, and the final memories are related by the global memory policy $\Gamma[\mathsf{entry}]$.

Let $\approx$ be the largest DND bisimulation.

Definition 2 (Delimited non-disclosure)**.** *A program P satisfies the delimited non-disclosure policy if* $\mathsf{entry} \approx \mathsf{entry}$.

The definition of delimited non-disclosure is termination-sensitive, in contrast to other declassification policies such as localised delimited release [4]. As it is usual in language-based security, the question of whether or not to use the termination-sensitive or termination-insensitive version of the security notion depends on the adversary model. Nevertheless, one could get a termination-insensitive version of DND by adding as hypothesis to the bisimulation that executions starting in s_i and t_j terminate.

We conclude this section with a brief remark about the nature of delimited non-disclosure. As announced, the policy is intended to capture declassification along the *what* and *where* dimensions. While the use of local policies clearly indicates that delimited non-disclosure supports the *where* aspect of declassification, it is not immediate from the definition to which extent the *what* dimension is supported. Clearly, local policies offer the opportunity to declassify variables, but do not explicitly mention the possibility of declassifying expressions. However, we are targeting unstructured languages in which intermediate computations are stored in intermediate memories, typically variables, and therefore it is sufficient to declassify variables instead of expressions. Furthermore, it is always possible to rewrite programs in a semantics-preserving fashion so that declassification of an expression e can be reduced to

declassification on a freshly introduced variable that stores the result of e. This is further elaborated in the next section, where we provide an example of this transformation.

4. Examples

In this section we introduce a simple sequential language to illustrate our policy and compare it with other declassification policies. To ease comparison with previous works where the local memory is inferred from the program syntax (see e.g. [12, 2, 4]), we consider a structured language, and include a syntactic construct for declassification.

Security lattice For the sake of simplicity, we work with a $L \leq H$ security lattice. Besides, we use h (resp. l) as a program variable whose initial memory policy is H (resp. L).

Language The syntax of the language is defined by the grammar in Figure 1 where n ranges over numbers $N = \{0, 1, \ldots\}$, x ranges over program variables, *op* ranges over arithmetic, boolean, and relational operators. As stated above, we include an explicit command for declassification, namely `declassify` (e) `in` $\{\ c\ \}$, to specify local policies by means of the program syntax.

In order to identify program points, some commands of the language are labelled with natural numbers i. We set $\mathsf{entry} = 1$. Local policies in our language might be directly specified by functions that map program points to memory policies. However, one can also choose to infer local memory policies from the program syntax, as explained in Example 1.

The language semantics is standard, and ommitted. The only non-standard command is `declassify` (e) `in` $\{\ c\ \}$—contrary to the introduction, we do not indicate the security level to which an expression is declassified, since there are only two levels. Informally, this command does not affect the semantics (its semantics is equivalent to a `skip`).

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid e \; op \; e \\
c & ::= & [\texttt{skip}]^i \mid c \,;\, c \\
 & & \mid [x := e]^i \\
 & & \mid [\texttt{if}\ (\ e\ )\ \texttt{then}\ \{\ c\ \}\ \texttt{else}\ \{\ c\ \}\ ]^i \\
 & & \mid [\texttt{while}\ (\ e\ )\ \texttt{do}\ \{\ c\ \}]^i \\
 & & \mid \texttt{declassify}\ (e)\ \texttt{in}\ \{\ c\ \}
\end{array}
$$

Figure 1. Expression and command syntax

The command `declassify` (e) `in` $\{\ c\ \}$ is used to specify a local policy where the security level of expression e is L in the scope of command c.

In order to capture the *what* dimension of declassification accurately, the command `declassify` (e) `in` $\{\ c\ \}$ declassifies whole expressions instead of single variables. DND can cope with this form of declassification using a simple program transformation, as is explained in Example 2.

Examples Our first example illustrates how local policies may be inferred from the syntax, and from the initial policy.

Example 1 (Local memory policy inference)**.** *Consider the program:*

$$[l_1 := 0]^1 \,;\, \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l_2 := h]^2\ \} \,;\, [l_3 := l_2]^3$$

For such a program, we generate local memory policies as follows:

$$
\begin{array}{l}
\Gamma[1](l_1) = \Gamma[1](l_2) = \Gamma[1](l_3) = L \\
\Gamma[1](h) = H \\
\Gamma[2](l_1) = \Gamma[2](l_2) = \Gamma[2](l_3) = L \\
\Gamma[2](h) = L \\
\Gamma[3] = \Gamma[1]
\end{array}
$$

Program point 3 *is not in the scope of the declassification command therefore its memory policy is the same of program point* 1.

This program complies with the DND policy for Γ *defined above because the direct flow from* h *to* l_2 *produced by assignment at* 2 *is authorised by the local memory policy* $\Gamma[2]$.

Our second example illustrates how delimited non-disclosure could capture declassification of expressions by an appropriate program transformation.

Example 2 (Declassification of expressions in DND)**.** *Expression level declassification can be accomplished in DND by assigning the declassified expression to a fresh variable and replacing expression occurrences by the new variable. For example, if we are interested in declassifying the expression* $h > 0$ *in the program:*

$$\texttt{declassify}\ (h > 0)\ \texttt{in}\ \{\ [\texttt{if}\ (\ h > 0\ )\ \texttt{then}\ \{\ [l := 0]^2\ \}]^1\ \}$$

we transform the program into:

$$
\begin{array}{l}
[h' := h > 0]^1 \,; \\
\texttt{declassify}\ (h')\ \texttt{in}\ \{\ [\texttt{if}\ (\ h'\ )\ \texttt{then}\ \{\ [l := 0]^3\ \}]^2\ \}
\end{array}
$$

where h' *is a fresh variable name, and generate the following memory policies:*

$$
\begin{array}{l}
\Gamma[1](l) = L \\
\Gamma[1](h) = \Gamma[1](h') = H \\
\Gamma[2](l) = \Gamma[2](h') = L \\
\Gamma[2](h) = H \\
\Gamma[3] = \Gamma[2]
\end{array}
$$

The next examples compare delimited non-disclosure with delimited release and its localised version.

Example 3 (Complies with DR and not DND)**.** *Consider the program:*

$$[l := h]^1 \ ; \ \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^2\ \}$$

The program does not comply with DND because command at program point 1 *contains an explicit flow for the initial local memory policy that says that* $\Gamma[1](l) = L$ *and* $\Gamma[1](h) = H$*. This program complies with* delimited release *[34], that only imposes restrictions on* what *is declassified without considering* where *declassification occurs. As discussed in [4] (p.1), a policy based only on the* what *dimension does not qualify for a declassification policy because it already assumes that secrets are known from the program's start. This program is not accepted by LDR.*

Example 4 (Complies with LDR and DND)**.** *Consider the program:*

$[h_2 := 0]^1$;
$[\texttt{if}\ (\ h_1\ )\ \texttt{then}\ \{\ \texttt{declassify}\ (h_1)\ \texttt{in}\ \{\ [l := h_1]^3\ \}\ \}\ \texttt{else}\ \{$
$\quad \texttt{declassify}\ (h_2)\ \texttt{in}\ \{\ [l := h_2]^4\ \}\ \}\ ]^2$

After the execution of this program the final value of l *is the value of* h_1 *even if* h_1 *has not been declassified. Delimited release accepts this program but DND and* localized delimited release *reject it.*

In contrast to delimited non-disclosure, localised delimited release rejects programs that may lead to laundering attacks [4] by allowing variables to be modified before their declassification. Indeed, our policy does not provide by itself any protection against laundering attacks.

We have essentially two reasons for not excluding laundering attacks by definition of DND. First of all, we believe that the indistinguishability of memories property (exclusively expressed by e.g. DND) and the lack of laundering attacks safety property are independent concepts. Hence there is no need to put both concepts together in a single property, as is the case in LDR. Furthermore, this well marked independence between the "indistinguishability" part of LDR and the laundering attack part of LDR leads as to conjecture that, given corresponding local memory policies, programs that comply with termination-sensitive version of LDR also comply with DND and programs that comply with DND and do not have laundering attacks, comply with LDR.

The second and most important reason is that by separating concepts of laundering attacks and DND, it is possible to modularly extend (as shown in Section 5) type systems for non-interference and even more important, construct a proof of soundness of this extension that can be adapted to a series of different languages, constructs, and non-interference type systems. Even more, laundering attacks can be avoided

$$\frac{}{\vdash_{LA} [\texttt{skip}]^i : \emptyset, \emptyset}$$

$$\vdash_{LA} [x := e]^i : \{x\}, \emptyset$$

$$\frac{\vdash_{LA} C_1 : U_1, V_1 \qquad \vdash_{LA} C_2 : U_2, V_2 \qquad U_1 \cap V_2 = \emptyset}{\vdash_{LA} C_1 \ ; \ C_2 : U_1 \cup U_2, V_1 \cup V_2}$$

$$\frac{\vdash_{LA} C_1 : U, V \qquad \vdash_{LA} C_2 : U, V}{\vdash_{LA} [\texttt{if}\ (\ e\ )\ \texttt{then}\ \{\ C_1\ \}\ \texttt{else}\ \{\ C_2\ \}\ ]^i : U, V}$$

$$\frac{\vdash_{LA} C : U, V \quad U \cap V = \emptyset}{\vdash_{LA} [\texttt{while}\ (\ e\ )\ \texttt{do}\ \{\ C\ \}]^i : U, V}$$

$$\frac{\vdash_{LA} C : U, V}{\vdash_{LA} \texttt{declassify}\ (x)\ \texttt{in}\ \{\ C\ \} : U, V \cup \{x\}}$$

$$\frac{\vdash_{LA} C : U, V \qquad U \subseteq U' \qquad V \subseteq V'}{\vdash_{LA} C : U', V'}$$

Figure 2. Effect system against laundering

by a simple effect system which ensures that variables that are declassified were not previously updated. To show this, we define a type system in Fig. 2 as an example of a type system to prevent laundering attacks. The judgements are of the form $\vdash_{LA} C : U, V$ where C is a command, U is a set of variables which meaning is variables that have been assigned by previous commands but not yet declassified and V is a set of variables which meaning is variables that have been declassified. The typing rules for composite commands, and in particular for loops, put disjointness constraints on the set of previously assigned variables and the set of declassified variables.

Example 5 (Laundering attacks)**.**

$$[h := 0]^1 \ ; \ \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^2\ \}$$

This program is rejected by localised delimited release and accepted by DND. However, the program is rejected by the laundering-attacks type system given in Fig. 2, since the only V *sets typing* $[h := 0]^1$ *must contain* h *by the rule of assingments, and the only* U *sets typing the declassify construct must contain* h *by the rule of declassification constructs. By the rule of sequential composition the* V *set of the assigment and the* U *set of the declassification instruction must not have common elements.*

Our next example suggests that it may be possible to encode localised delimited release using DND. We conjecture that a terminating program is accepted by localized delimited release iff its transformation along the process de-

scribed above is accepted by delimited non-disclosure and by the effect system against laundering.

Example 6 (Program complies with LDR but not with DND)**.** Localised delimited release *accepts programs like:*

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^1\ ;\ [l_2 := h]^2\ \}\ ;\ [l_3 := h]^3$$

because it considers that assignment to l_3 is "intuitively" secure because the program can be safely transformed into:

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^1\ ;\ [l_2 := h]^2\ \}\ ;\ [l_3 := l_2]^3$$

For DND, assignments 1 and 2 are valid because they are made in the scope of the declassification of variable h, but assignment 3 does not comply with our policy.

Notice that, if necessary, the original program can be transformed to be accepted by DND by extending the declassification region until the end of the program:

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^1\ ;\ [l_2 := h]^2\ ;\ [l_3 := h]^3\ \}$$

We conclude this section by a comparison with non-disclosure, which is closely related to DND. The main difference is that DND permits a fine-grained relaxation of non-interference by authorising exceptional flows not from *all* high variables but from user defined sets of *high* variables. In contrast, non-disclosure is based on a global security lattice G that dynamically evolves to interpret locally induced flow policies. These policies introduce new flow relations in G in the context of sets of program points. Outside these sets, the global security lattice remains as it was at the beginning of the program.

The following example compares non-disclosure [2] and delimited non-disclosure, and suggests a possible encoding of ND in terms of DND. In contrast to non-disclosure, we assume that the security lattice does not change at different points of the program but variable confidentiality levels change. Thus, our security policy is not expressed by means of flows between principals (that determine the security lattice) but rather by local memory policies.

Example 7 (Non-disclosure)**.** *The flow declaration construct of non-disclosure*

$$\texttt{flow}\ (\mathtt{p}_2 \prec \mathtt{p}_1)\ \texttt{in}\ c$$

is introduced to express local lattice modifications. Here c is a statement into which the original security lattice induced by principals $\{p_2, p_1\}$ is modified to permit flows from p_2 to p_1. Let's name elements in the original lattice as

$$H = \{\}$$
$$H_1 = \{p_1\}$$
$$H_2 = \{p_2\}$$
$$L = \{p_1, p_2\}$$

where $\leq$ is given by the subset relation. The new lattice induced by the flow construct above, treats H_2 as L (security level H_2 dissapears from the lattice).

In our policy the security lattice is not modified, instead the initial memory policy $\Gamma[1]$ is adapted at each program point in order to reflect local relaxations of non-interference.

Therefore, the effect of the above flow construct, in terms of our policy, is that for all variable x such that $\Gamma[1](x) = H_2$ we have $\Gamma[j](x) = L$ if j is a program point for statement c.

For example, if the program has variables h and h' with $\Gamma[1](h) = \Gamma[1](h') = H_2$, then $\texttt{flow}\ (\mathtt{p}_2 \prec \mathtt{p}_1)\ \texttt{in}\ c$ can be expressed in our language as:

$$\texttt{declassify}\ (h)\ \texttt{in}\ \{\ \texttt{declassify}\ (h')\ \texttt{in}\ \{\ c\ \}\ \}$$

The flow construct of non-disclosure is expressed in our setting as a declassification of all variables belonging to the security levels induced by the new flow.

5. Enforcing delimited non-disclosure

The purpose of this section is to provide a modular definition and soundness proof of an information flow type system that enforces delimited non-disclosure. For simplicity, we fix the lattice of security levels to $\mathcal{S} = \{L, H\}$ with $L \leq H$ and assume that security policies are functions that attach security levels to program variables. (However the main result applies to arbitrary lattices, the simplification to two level lattices applies to the hypotheses on NI, and these hypotheses can be generalized as shown in e.g. [6].)

The order on security levels can be extended pointwise to security policies; by abuse of notation, we let $\leq$ denote the order between policies.

5.1 Assumptions on NI typing

Our starting point is a type system $\vdash_{NI}$ designed for enforcing non-interference. The type system operates on a program P annotated with control dependence regions $(\mathsf{region}, \mathsf{junction})$, and is parameterised by a security environment se that maps program points to levels, a policy Γ that maps variables to security levels, and a type S; the exact nature of types does not need to be specified. For readability, typing judgements are written in the following form:

$$\Gamma, S \vdash_{NI} i$$

where Γ is a policy, S is a type, i is a program point. All other parameters are left implicit.

The principal hypotheses that we make on $\vdash_{NI}$ take the form of unwinding statements. The first unwinding hypothesis states that execution locally respects state equivalence, i.e. if we start from two equivalent states that point to the same instruction and perform one step of execution, then the two resulting states are also equivalent. Additionally, it

makes some technical assumption about the program counters of the resulting states: either they coincide, or the initial states were pointing to a high branching instruction.

In order to formulate the notion of high branching instruction, we say that k has a high region, written $\mathsf{highregion}(k)$, iff k is a branching point, and $se(j) = H$ for every $j \in \mathsf{region}(k)$. Moreover, we write $i \in \mathsf{highregion}(k)$ to mean $\mathsf{highregion}(k)$ and $i \in \mathsf{region}(k)$.

Hypothesis 4 (LowNI)**.** *Assume that* $\Gamma, S \vdash_{NI} i$*, and* $s_i \sim_\Gamma t_i$*, and* $s_i \rightsquigarrow s'_{i'}$ *and* $t_i \rightsquigarrow t'_{j'}$*. Then* $s'_{i'} \sim_\Gamma t'_{j'}$ *and one of the following holds:*

- $i' = j'$*, or*
- $\mathsf{highregion}(i)$*, and* $i', j' \in \mathsf{region}(i) \cup \{\mathsf{junction}(i)\}$.

The second hypothesis states that executing an instruction in a high control dependence region does not modify the observable part of a state. It corresponds to the step preserving unwinding property of [30].

Hypothesis 5 (HighNI)**.** *Assume that* $\mathsf{highregion}(k)$ *and let* $i, i' \in \mathsf{region}(k)$*. Assume that* $s_i \rightsquigarrow s'_{i'}$ *and* $\Gamma, S \vdash_{NI} i$*. Then* $s_i \sim_\Gamma s'_{i'}$.

Using the above lemmas, one can prove that typable programs are non-interfering.

Definition 3. *A program P is typable with type S w.r.t. Γ, written $\Gamma, S \vdash_{NI} P$, iff for every program point i, we have $\Gamma, S \vdash_{NI} i$.*

Using an adaptation of the general scheme of [7], one can prove that, under the hypotheses of this section, typable programs are non-interfering. More precisely, if P is typable w.r.t. Γ, then P is non interferent w.r.t. Γ, i.e. for all $s_{\mathsf{entry}}, t_{\mathsf{entry}}$:

$$\left.\begin{array}{l} s_{\mathsf{entry}} \sim_\Gamma t_{\mathsf{entry}} \\ s_{\mathsf{entry}} \rightsquigarrow^\star s'_i \\ t_{\mathsf{entry}} \rightsquigarrow^\star t'_j \\ i, j \in \mathcal{P}_{\mathrm{exit}} \end{array}\right\} \Rightarrow s'_i \sim_\Gamma t'_j$$

5.2 Typing delimited non-disclosure

The goal of this section is to formulate the DND-type system and to prove that, under some mild hypotheses, programs that are typable by the DND-type system verify the delimited non-disclosure policy. We begin by defining the type system.

The type system for delimited non-disclosure is very similar to the type system for non-interference, but is parameterised by a family of policies $(\Gamma[i])_{i\in\mathcal{P}}$. Like the type system for non-interference, the type system for delimited non-disclosure is parameterised by control dependence regions $(\mathsf{region}, \mathsf{junction})$, a security environment se and a type S.

Definition 4. *Let $(\Gamma[i])_{i\in\mathcal{P}}$ be an indexed set of local policies. A program P is typable with type S w.r.t. $(\Gamma[i])_{i\in\mathcal{P}}$, written $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$, iff for every program point i, we have $\Gamma[i], S \vdash_{NI} i$.*

The DND type system is conservative: intuitively, programs without declassification (i.e. without different local policies) that are typable by the DND type system are non-interferent programs. However, $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$ does not necessarily imply that $\Gamma[i], S \vdash_{NI} P$ for some $i \in \mathcal{P}(P)$.

Note that, at such an abstract level, there is a strong similarity between our type system for delimited non-disclosure, and flow-sensitive type systems [20], since they both rely on a family of policies $(\Gamma[i])_{i\in\mathcal{P}}$. However, the type system for delimited non-disclosure makes different assumptions on this family: see Hypothesis 7.

Termination Delimited non-disclosure is termination sensitive and the type system must reject any program that has loops whose termination behaviour is influenced by confidential data. Therefore, we must of a program analysis loop that detects that the branching point i is a loop, and we assume that the type system guarantees that all high loops terminate. In the sequel, we write $\mathsf{loop}(i)$ if the analysis detects that i is a loop, see e.g. [28] for a definition of such an analysis.

Hypothesis 6 (Termination of while loops)**.** *Assume* $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$*, and let i be a program point such that* $\mathsf{highregion}(i)$ *and* $\mathsf{loop}(i)$*. Then* $\mathsf{junction}(i)$ *is defined. Besides, if* $j \in \mathsf{region}(i)$*, and s_j is safe, i.e.* $\vdash_{\mathrm{safe}} s_j$*, then there exists $t_{j'}$ such that $s_j \rightsquigarrow^\star t_{j'}$, and $j' = \mathsf{junction}(i)$.*

The hypothesis states that high loops terminate normally (in contrast to abrupt termination caused by a return in a loop). This condition can be brutally enforced by typing rules that reject all high loops [36, 2]. However, Gérard Boudol [11] observed that such typing rules can be largely improved by being parameterised by a termination analysis (see e.g. [15] for an advanced analysis of this kind).

Note that one can strengthen the hypothesis by not requiring that i is a loop, using Hypothesis 2.

Lemma 1 (Exit from high guards)**.** *Assume* $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$*, and let i be a program point such that* $\mathsf{highregion}(i)$ *and* $\mathsf{junction}(i)$ *is defined. If* $j \in \mathsf{region}(i)$*, and s_j is safe, i.e.* $\vdash_{\mathrm{safe}} s_j$*, then there exists $t_{j'}$ such that $s_j \rightsquigarrow^\star t_{j'}$, and $j' = \mathsf{junction}(i)$.*

Proof. If $\mathsf{loop}(i)$, then we apply directly Hypothesis 6. Otherwise, one can apply Hypotheses 1 and 2, together with progress (Hypothesis 3). □

We use Lemma 1 in the proof of Theorem 1 below.

Correct memory policies In order to be able to prove soundness of the DND type system, we impose mild restrictions on families of local memory policies $(\Gamma[i])_{i\in\mathcal{P}}$. These restrictions can be verified automatically independently of the DND type system.

Hypothesis 7 (Correct memory policies)**.** *A family of memory policies $(\Gamma[i])_{i\in\mathcal{P}}$ is correct if for all $i, j \in \mathcal{P}$:*

1. *for every variable x, we have $\Gamma[i](x) \leq \Gamma[\mathsf{entry}](x)$;*
2. *if $\mathsf{highregion}(i)$ and $j \in \mathsf{region}(i) \cup \{\mathsf{junction}(i)\}$, then $\Gamma[j] = \Gamma[i]$.*

The first item says that a memory policy for a variable x can be downgraded (declassified) but not upgraded. The second item says that inside high control dependence region and up to the junction point, local policies remain unchanged; this assumption is in line with previous work on disallowing declassification inside high branching statements, and rightfully rejects programs such as

$$[\texttt{if}\ (\ h\ )\ \texttt{then}\ \{\ \texttt{declassify}\ (h_1)\ \texttt{in}\ \{\ [l := h_1]^2\ \}\ \}]^1$$

which leaks the value of h through the knowledge of whether h_1 has been declassified or not (see Example 4).

Hypothesis 8 (Monotonicity of $\sim$)**.** *If $\Gamma[i] \leq \Gamma[j]$ and $s \sim_{\Gamma[i]} t$ then $s \sim_{\Gamma[j]} t$.*

The hypothesis guarantees monotonicity of $\sim_{\Gamma[i]}$ w.r.t. the order of local memory policies.

Soundness proof To prove that the DND type system enforces delimited non-disclosure, we exhibit a DND bisimulation $\mathcal{B}$ that satisfies $\mathsf{entry}\ \mathcal{B}\ \mathsf{entry}$. The relation $\mathcal{B}$ is defined inductively by the clauses

$$\frac{}{i\ \mathcal{B}\ i} \qquad \frac{j\ \mathcal{B}\ i}{i\ \mathcal{B}\ j}$$

$$\frac{i, j \in \mathsf{highregion}(k)}{i\ \mathcal{B}\ j}$$

$$\frac{i \in \mathsf{highregion}(k) \quad j = \mathsf{junction}(k)}{i\ \mathcal{B}\ j}$$

Since $\mathcal{B}$ is reflexive, we obviously have $\mathsf{entry}\ \mathcal{B}\ \mathsf{entry}$.

In the sequel, we assume that all aforementioned hypotheses are satisfied.

Theorem 1 (Soundness of $\vdash_{DND}$)**.** *If $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$, then P complies with the delimited non-disclosure policy w.r.t. $(\Gamma[i])_{i\in\mathcal{P}}$.*

Proof. We show that $\mathcal{B}$ is a DND bisimulation. Assume that $i\mathcal{B}j$. There are four cases to treat:

- If $i = j$. Let s_i and t_i be states s.t. $s_i \rightsquigarrow s'_{i'}$ and $s_i \sim_{\Gamma[i]} t_i$. Suppose that $\mathrm{safe}_P(t_j)$. By progress (Hypothesis 3), either $i \in \mathcal{P}_{\mathrm{exit}}$ or there exists $t'_{j'}$ such that $t_i \rightsquigarrow t'_{j'}$. Since $s_i \rightsquigarrow s'_{i'}$, we have $i \notin \mathcal{P}_{\mathrm{exit}}$, and thus there exists $t'_{i'}$ such that $t_i \rightsquigarrow t'_{j'}$. By locally respects unwinding (Hypothesis 4), $s'_{i'} \sim_{\Gamma[i]} t'_{j'}$ and $i' = j'$, or $i', j' \in \mathsf{region}(i) \cup \{\mathsf{junction}(i)\}$. In all cases, we have $i'\mathcal{B}j'$.

  Furthermore, by Hypothesis 7 $\Gamma[i] \leq \Gamma[\mathsf{entry}]$ and by Hypothesis 8 $s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'}$, so we are done.

- If $i, j \in \mathsf{highregion}(k)$ for some k. Let $s_i \sim_{\Gamma[i]} t_j$, and assume that $s_i \rightsquigarrow s'_{i'}$. By step preserving unwinding (Hypothesis 5), $s'_{i'} \sim_{\Gamma[i]} t_j$. By monotonicity of local policies (Hypothesis 7), $s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_j$. Furthermore, exit through junction (Hypothesis 1) ensures that $\mathsf{junction}(i)$ is the unique exit point of $\mathsf{region}(i)$, therefore either $i' \in \mathsf{region}(k)$ or $i' = \mathsf{junction}(k)$. In both cases, $i'\mathcal{B}j$.

- If $i \in \mathsf{highregion}(k)$ and $j = \mathsf{junction}(k)$ for some k. This case is similar to the above (except for the fact that if $i' = \mathsf{junction}(k)$ we use the reflexivity of $\mathcal{B}$ to conclude that $i'\mathcal{B}j$).

- If $j \in \mathsf{highregion}(k)$ and $i = \mathsf{junction}(k)$ for some k. Let $s_i \sim_{\Gamma[i]} t_j$, and assume that $s_i \rightsquigarrow s'_{i'}$. By progress (Hypothesis 3) and exit from high guards (Lemma 1), and by exit through junction (Hypothesis 1), there exists a sequence

  $$t_j \rightsquigarrow u^1_{k_1} \rightsquigarrow \ \ldots\ \rightsquigarrow u^l_{k_l} \rightsquigarrow u'_i$$

  such that $k_1 \ldots k_l \in \mathsf{region}(i)$. By repeatedly applying the step preserves unwinding (Hypothesis 5), appealing to the correctness of memory policy (Hypothesis 7), which ensures that policy do not vary in high regions, and the transitivity of state equivalence, we conclude that $t_j \sim_{\Gamma[k]} u'_i$. Since $\Gamma[k] = \Gamma[i]$ by correctness of memory policy (Hypothesis 7), we have by transitivity $s_i \sim_{\Gamma[i]} u'_i$, and can conclude as in the first case.

□

6. Case study: Java Virtual Machine

The objective of this section is to apply our results to a minimal fragment of the JVM. We also establish type-preserving compilation w.r.t. a type system for the language of Section 4. Finally, we discuss the applicability of the method to a larger fragment of the JVM.

$instr$	::=	binop op	binary operation on stack
	|	push v	push value on top of stack
	|	load x	load value of x on stack
	|	store x	store top of stack in variable x
	|	ifeq j	conditional jump
	|	goto j	unconditional jump

Figure 3. $\mathrm{JVM}_{\mathcal{I}}$ **instructions**

6.1 Language and policy

For brevity, we consider a fragment called $\mathrm{JVM}_{\mathcal{I}}$, whose instruction set is given in Figure 3; we use op to range over binary operations, v over values, x over variables, and j over program points.

The operational semantics of $\mathrm{JVM}_{\mathcal{I}}$ programs is standard, and given by a small-step relation $\rightsquigarrow$ that represents one step execution of the virtual machine. States can either be intermediate, in which case they consist of an operand stack, a memory, and a program counter, or final, in which case they consist of a memory. We use $\langle i, \rho, os\rangle$ to denote an intermediate state with program counter i, memory ρ and operand stack os. Final states are simply identified with memories.

To instantiate delimited non-disclosure to $\mathrm{JVM}_{\mathcal{I}}$ programs, we must first define local policies. A local policy is simply a mapping from variables to levels. We assume given a policy $\Gamma[i]$ for each program point i.

Next, we define an indexed family of partial equivalence relations between states. This involves defining equivalence between memories, and between operand stacks.

Definition 5. *Two memories μ and μ' are equivalent w.r.t. Γ, written $\mu \sim^{\mathsf{Mem}}_{\Gamma} \mu'$, iff $\mu(x) = \mu'(x)$ for every variable x such that $\Gamma(x) = L$.*

Equivalence between operand stacks is defined relative to stack types. (There are both weaker and stronger notions of operand stack equivalence; see [7] for a discussion on these notions).

Definition 6. *The relation $os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2$, where $st_1, st_2 \in \mathcal{S}^*$, is defined inductively, together with the inductively defined auxiliary relation $\mathsf{high}(os, st)$, in Figure 4.*

Finally, state equivalence is defined in the obvious way.

Definition 7. *Let $S : \mathcal{P} \to \mathcal{S}^*$. Two states $s = \langle i, \rho, os\rangle$ and $s' = \langle i', \rho', os'\rangle$ are equivalent, written $s \sim^{\mathsf{State}} s'$, iff $\Gamma(i) = \Gamma(i')$ and $\rho \sim^{\mathsf{Mem}}_{\Gamma(i)} \rho'$ and $os \sim^{\mathsf{Stk}}_{S(i),S(i')} os'$.*

To conclude with the definition of delimited non-disclosure, one needs to define the notion of safe state. In our setting, a safety type assigns to each program point a natural number that represents the height of its operand stack, and a state is safe if its operand stack has the correct height w.r.t. its program counter. The safety type system tracks the height of the operand stack, and ensures that jumps are correct, i.e. remain within the program code. It is easy to show Hypothesis 3, i.e. that safe states enjoy progress. In a more general setting, one can define safe states using the work of Freund and Mitchell [16], who formalized a safety type system for the JVM, and showed that safe programs enjoy progress.

$$\frac{\mathsf{high}(os_1, st_1) \qquad \mathsf{high}(os_2, st_2)}{os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2}$$

$$\frac{os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2}{v :: os_1 \sim^{\mathsf{Stk}}_{L::st_1,L::st_2} v :: os_2}$$

$$\frac{os_1 \sim^{\mathsf{Stk}}_{st_1,st_2} os_2}{v_1 :: os_1 \sim^{\mathsf{Stk}}_{H::st_1,H::st_2} v_2 :: os_2}$$

$$\frac{}{\mathsf{high}(\epsilon, \epsilon)} \qquad \frac{\mathsf{high}(os, st)}{\mathsf{high}(v :: os, H :: st)}$$

Figure 4. Operand stack equivalence

6.2 Type system

The type system is expressed by rules of the form

$$i \overset{JVM}{\vdash} st \Rightarrow st'$$

where i is a program point and $st, st' \in \mathcal{S}^\star$ are stacks types. The rules are given in Figure 5, and assume that programs come equipped with control dependence regions (cdr), and a security environment. The rules exactly match the rules of [6], except that:

- the rules for load and store use the local policy;
- the rule for ifeq rejects high loops, and is instantiated to the case where the stack is empty after execution (which is the case for compiled programs, see [23]).

The typing rules of $\mathrm{JVM}_{\mathcal{I}}$ can be viewed as an instance of the generic type system. Indeed, define a type to be a map S from program points to stack types. Then, we define $(\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P$ iff the following holds:

- $S(\mathsf{entry})$ is the empty stack;
- for every i s.t. $se(i) = H$, the stack $S(i)$ is high (i.e. all elements of $S(i)$ are equal to H);

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash_{DND} st \Rightarrow se(i) :: st}$$

$$\frac{P[i] = \mathsf{binop}\ op}{i \vdash_{DND} k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2) :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \Gamma_i(x)}{i \vdash_{DND} k :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash_{DND} st \Rightarrow (\Gamma_i(x) \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash_{DND} st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{return} \qquad se(i) = L}{i \vdash_{DND} k :: st \Rightarrow \epsilon}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \mathsf{loop}(i) \Rightarrow k = L \quad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash_{DND} k :: \epsilon \Rightarrow \epsilon}$$

Figure 5. Transfer rules for $\mathrm{JVM}_{\mathcal{I}}$ instructions

- if $i \mapsto j$, then $i \overset{JVM}{\vdash} S(i) \Rightarrow st$ for some $st \leq S(j)$.

Under this definition, and restricting ourselves to constant families of policies (i.e. families of policies $(\Gamma[i])_{i\in\mathcal{P}}$ s.t. $\Gamma[i] = \Gamma[j]$ for all i and j), the notion of typable program w.r.t. $\vdash_{NI}$ coincides with the notion of typable program in [6]. Furthermore, one can use our construction of $\vdash_{DND}$, Theorem 1 and our earlier results in the proof of non-interference for $\mathrm{JVM}_{\mathcal{I}}$ to conclude that $\vdash_{DND}$ enforces delimited non-disclosure.

Theorem 2. *Let $(\Gamma_i)_{i\in\mathcal{P}}$ be correct policies. Let P be a safe program such that $(\Gamma_i)_{i\in\mathcal{P}}, S \vdash_{\mathrm{DND}} P$. If* (region, junction) *satisfy the SOAP properties (given in Figure 6), then P satisfy delimited non-disclosure.*

We briefly indicate why the hypotheses of Section 5 hold. Exit through junction (Hypothesis 1) corresponds exactly to the property **SOAP2**, whereas no return before junction (Hypothesis 2) corresponds exactly to the property **SOAP3**.

Progress and preservation of safety (Hypothesis 3) hold as explained above.

The unwinding statements (Hypotheses 4 and 5) are direct consequences of the unwinding lemmas proved in [6]; note that the unwinding statements are proved using the **SOAP** properties.

Hypothesis 6 holds by definition of the typing rule for ifeq, which prevents highregion(i) and loop(i) from holding simultaneously.

SOAP1 for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in \mathsf{region}(i)$ or $k = \mathsf{junction}(i)$;

SOAP2 for all program points i, j, k, if $j \in \mathsf{region}(i)$ and $j \mapsto k$, then either $k \in \mathsf{region}(i)$ or $k = \mathsf{junction}(i)$;

SOAP3 for all program points i, j, if $j \in \mathsf{region}(i)$ and $j \in \mathcal{P}_{\mathrm{exit}}$ then $\mathsf{junction}(i)$ is undefined.

Figure 6. SOAP properties

Finally, correctness of memories (Hypothesis 7) is an assumption of the theorem, and monotonicity (Hypothesis 8) holds trivially.

6.3 Type-preserving compilation

In this section, we focus on preservation of typability by compilation. The benefits of type preservation are two-fold: they guarantee program developers that their programs written in an information flow aware programming language will be compiled into executable code that will be accepted by a security architecture that integrates an information flow bytecode verifier. Conversely, they guarantee code consumers of the existence of practical tools to develop applications that will provably meet the policy enforced by their information flow aware security architecture.

We consider the source language introduced in Section 4, and define a declassification type system. The type system is parameterized by a family $(\Gamma[i])_i$ of local policies, in this case one policy per label. As already explained in Example 1, these local policies can be inferred from the initial policy, and from the program syntax. (Alternatively, one could formulate a type system that is parameterized by a single policy, that corresponds to the policy at the entry point of the program, and use the type system to track the local changes in the policy.)

Furthermore, the local policies $(\Gamma_i)_{i\in\mathcal{P}}$ for $T(P)$ are generated from the initial policy of P (that is the policy of the entry point of P) and from the `declassify` (.) `in` { . } constructs, as explained in Example 1.

The type system for the source language is given by the rules given in Figs. 7 and 8. Notice that since we have local policies $(\Gamma[i])_i$ for each program point, the rule for declassification corresponds exactly to typability of C and the type system is very similar to a non-interference type system except because it uses a set of local policies instead of a unique global policy.

Notice furthermore that the typing rules are restricted to programs in which only variables are declassified. In order to extend typing to programs that do not meet this restriction, we use the source-to-source trans-

$$\frac{}{(\Gamma[i])_i \vdash_{DND} n : L}$$

$$\frac{}{(\Gamma[i])_i \vdash_{DND} x : \Gamma(x)}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e_1 : k \qquad (\Gamma[i])_i \vdash_{DND} e_2 : k}{(\Gamma[i])_i \vdash_{DND} e_1 \; op \; e_2 : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : k_1 \qquad k_1 \leq k_2}{(\Gamma[i])_i \vdash_{DND} e : k_2}$$

Figure 7. Typing rules for expressions

$$\frac{}{(\Gamma[i])_i \vdash_{DND} [\texttt{skip}]^i : L}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : \Gamma_i(x)}{(\Gamma[i])_i \vdash_{DND} [x := e]^i : \Gamma(x)}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : k \qquad (\Gamma[i])_i \vdash_{DND} C_1 : k \qquad (\Gamma[i])_i \vdash_{DND} C_2 : k}{(\Gamma[i])_i \vdash_{DND} [\texttt{if} \; (\; e \;) \; \texttt{then} \; \{ \; C_1 \; \} \; \texttt{else} \; \{ \; C_2 \; \}]^i : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} C : k}{(\Gamma[i])_i \vdash_{DND} \texttt{declassify} \; (x) \; \texttt{in} \; \{ \; C \; \} : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} e : L \qquad (\Gamma[i])_i \vdash_{DND} C : L}{(\Gamma[i])_i \vdash_{DND} [\texttt{while} \; (\; e \;) \; \texttt{do} \; \{ \; C \; \}]^i : k}$$

$$\frac{(\Gamma[i])_i \vdash_{DND} C : k \qquad k' \leq k}{(\Gamma[i])_i \vdash_{DND} C : k'}$$

Figure 8. Typing rules for commands

formation introduced in Example 2. This transformation replaces `declassify` (e) `in` $\{ \; c \; \}$ by $x := e;$ `declassify` (x) `in` $\{ \; c' \; \}$, where x is a fresh variable and c' is recursively obtained from applying the same transformation to $c[e/x]$. This transformation is semantics-preserving provided variables in e are not modified in c. In the sequel, we denote by $T(P)$ the result of applying this transformation to P.

We consider a non-optimizing compiler $[\![.]\!]$. Its definition on programs is standard, except for the statement `declassify` (x) `in` $\{ \; c \; \}$, which is compiled to $[\![c]\!]$ (that is, declassify statements are ignored by compilation). The compiler is extended to programs that delassify expressions by composition with the transformation T.

As the bytecode type system uses both a cdr structure $(\mathsf{region}, \mathsf{junction})$, a security environment se, and local policies $(\Gamma[i])_{i \in \mathcal{P}}$, the compiler must also generate this additional information. Furthermore, the generated information must ensure that $[\![P]\!]$ is typable w.r.t. $(\mathsf{region}, \mathsf{junction})$, $(\Gamma[i])_{i \in \mathcal{P}}$ and se. The cdr structure and security environment of the compiled programs can be defined as in earlier works on type-preserving compilation, e.g. [8].

The compiler maps every labeled statement in the source programs to a set of program points. The local policy of these program points is inherited from the local policy of the label of their corresponding source statement.

Theorem 3 (Typability Preservation)**.** *Let P be a source program with correct memory policies. Assume that $T(P)$ is typable by the DND source type system. Then $[\![P]\!]$ is a typable bytecode program (w.r.t. the generated information).*

In addition, the generated cdr structure $(\mathsf{region}, \mathsf{junction})$ satisfies the SOAP properties and the generated local policies $(\Gamma[i])_{i \in \mathcal{P}}$ are correct. Therefore, one can conclude by Theorem 2 that the compiled program verifies delimited non-disclosure w.r.t. the family of local policies generated by the compiler.

Section 4 also presents an effect system to prevent laundering attacks. Although we refrain from doing so here, it is possible to define a similar effect system for the $\mathrm{JVM}_{\mathcal{I}}$ and show that compilation preserves typability wr.t. this system.

7 Discussion

7.1 Objects, exceptions, and methods

Our method has been described and instantiated in a representative, but simplified, setting. Leveraging it to the sequential fragment of the Java Virtual Machine does not pose any major difficulty, but involves a significant amount of technicalities. Fortunately, these technicalities were already handled in the NI work on the JVM.

The first class of technicalities arises from dealing with object-oriented features. Firstly, information flow type systems for the JVM must rely on security signatures with exception effects to support modular verification, and therefore to remain compatible with bytecode verification. Furthermore, signatures and specifications must be compatible with method overriding. Secondly, in presence of objects, state equivalence is formulated in terms of heap equivalence, which must be carefully handled to avoid flows based on non-opaqueness of pointers [19]. Thirdly, exceptions introduce some additional potential sources of indirect flows, and thus must be accounted for in the type system.

In addition, further technicalities are required to achieve an analysis with sufficient precision. Indeed, the presence of exceptions and object-orientation yields a significant blow-up in the control flow graph of the program, and, if no care is taken, may lead to overly conservative type-based analyses. In order to achieve an acceptable degree of usability,

the information flow type system of [6] relies on preliminary analyses that provide a more accurate approximation of the control flow graph of the program. Typically, the preliminary analyses will perform safety analyses such as class analysis, null pointer analysis, exception analysis, and array out-of-bounds analysis. These analyses drastically improve the quality of the approximation of the control flow graph. In particular, one can define a tighter successor relation $\mapsto$ that leads to more precise control dependence regions; in [6], precision is further increased by indexing $\mapsto$ and control dependence regions by a tag (an exception or a special tag for normal execution).

7.2 Multi-threading

As multi-threading is widely used in applications to mobile code, there is a strong interest in developing enforcement mechanisms for multi-threaded programs. There is a wide range of works that consider information flow policies for multi-threaded source programs, see e.g. [10, 25, 32, 33], and it would be interesting to understand how the modular technique of this paper could be applied to this setting.

Building upon earlier work by Russo and Sabelfeld [31], [9] considers a modular method to devise sound enforcement mechanisms for multi-threaded programs. The central idea of these works is to constrain the behavior of the scheduler so that it does not leak information; it is achieved by giving to the scheduler access to the security levels of program points, and by requiring that the choice of the thread to be executed respects appropriate conditions. As in the present paper, the type system for the concurrent language is defined in a modular fashion from the type system for the sequential language, and the soundness of the concurrent type system is derived from unwinding lemmas for the sequential type system.

The kind of extension presented here is orthogonal to the multi-threaded extension shown in [9], and we believe that modular extensions can be combined to augment policy and language expressivity in a single bytecode verifier that enforces DND for a concurrent JVM. Understanding the intuitive guarantees provided by the extension of DND to concurrent languages and formalizing the details of the combination of [9] with the results of this paper is left for future work.

7.3 Formal proofs

Information flow type systems are complex mechanisms whose soundness proofs are particularly involved, especially when considering permissive declassification policies for real programming languages such as the JVM. As such type systems are designed to complement existing type systems for safety and thus lie at the heart of the Trusted Computing Base (TCB), it is therefore fundamental that their implementation is correct, since flaws in the implementation of a type system can be exploited to launch attacks. In our earlier work [6], we have used the proof assistant Coq to formally verify the soundness of an information flow type system that ensures non-interference for a sequential fragment of the Java Virtual Machine. In addition to providing strong guarantees about the correctness of the type system, the formalization serves as a basis for a Foundational Proof Carrying Code architecture. A distinctive feature of our architecture is that the type system is executable inside higher order logic and thus one can use reflection for verifying certificates within Coq, or extraction to obtain an OCaml implementation of a lightweight information flow checker. As compared to Foundational Proof Carrying Code [3], which is deductive in nature, reflective Proof Carrying Code exploits the interplay between deduction and computation to support efficient verification procedures and compact certificates.

As a benefit of the modularity of our approach, we believe that it is possible to achieve, at a moderate cost, a proof of soundness for our DND information flow type system presented, using the formalization reported in [6]. To be more specific, the Coq development is organized in two parts: a generic part, that derives the soundness of the non-interference type system from the unwinding lemmas, and a specific part, that establishes the unwinding lemmas for a particular language, operational semantics, and type system. We are confident that extending the generic part of the formalization to accomodate DND is direct, as in fact proving Theorem 1 in an abstract setting is direct. Nevertheless, we anticipate a fair amount of bookkeeping in the instantiation: even if there is no conceptual difficulty in programming in Coq a bytecode verifier that enforces DND, the specification of the non-interference type system for the JVM is rather large, and extending (even in the modular fashion) its definition to DND—and thus to have a policy per program point instead of a global policy—will be demanding.

8 Conclusion

Tractable enforcement of declassification policies for bytecode languages is an essential step towards a practical use of language-based security in mobile code. In this paper, we have developed a modular method to extend non-interference type systems to sound information flow type systems for delimited non-disclosure, a security policy that combines the *what* and *where* dimensions of declassification, and that is closely related to policies such as delimited release, localized delimited release, and non-disclosure. As a case study, we have instantiated our results to a sequential

fragment $\mathrm{JVM}_{\mathcal{I}}$ of the Java Virtual Machine, yielding the first sound information flow type system to support declassification for an unstructured language. In addition, we have argued that our approach is scalable to exceptions, objects, and methods. As a final contribution, we have shown that our results on type-preserving compilation readily adapt to declassification.

As future work, we intend to spell out the details of extening our results to a richer language with object-oriented features and concurrency, and to provide machine-checked proofs of our results.

Acknoledgements: We thank Ana Almeida Matos, Gérard Boudol, and anonymous reviewers for providing insightful comments on the final version of this paper. This work is partially funded by the EU project MOBIUS and by the ANR project PARSEC.

References

[1] *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society, 2005.

[2] A. Almeida Matos and G. Boudol. On Declassification and the Non-Disclosure Policy. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

[3] A. W. Appel. Foundational Proof-Carrying Code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247, Washington, DC, USA, 2001. IEEE Computer Society.

[4] A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, New York, NY, USA, 2007. ACM.

[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *29th IEEE Symposium on Security and Privacy*, May 2008.

[6] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In Nicola [27], pages 125–140.

[7] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. Technical report, INRIA, 2007. Extended version of [6]. Available from `http://hal.inria.fr/inria-00106182/`.

[8] G. Barthe, T. Rezk, and D. A. Naumann. Deriving an Information Flow Checker and Certifying Compiler for Java. In *27th IEEE Symposium on Security and Provacy*, pages 230–242. IEEE Computer Society, 2006.

[9] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of Multithreaded Programs by Compilation. In J. Biskup and J. Lopez, editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007.

[10] A. Bossi, C. Piazza, and S. Rossi. Compositional information flow security for concurrent programs. *Journal of Computer Security*, 15(3):373–416, 2007.

[11] G. Boudol. On Typing Information Flow. In D. V. Hung and M. Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2005.

[12] N. Broberg and D. Sands. Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In P. Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2006.

[13] S. Chong and A. C. Myers. Language-based information erasure. In aa [1], pages 241–254.

[14] S. Chong and A. C. Myers. End-to-end enforcement of erasure. In *CSFW*. IEEE Computer Society, 2008. To appear.

[15] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.

[16] S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *J. Autom. Reason.*, 30(3-4):271–321, 2003.

[17] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In N. D. Jones and X. Leroy, editors, *POPL*, pages 186–197. ACM, 2004.

[18] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[19] D. Hedin and D. Sands. Noninterference in the Presence of Non-Opaque Pointers. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 217–229, Washington, DC, USA, 2006. IEEE Computer Society.

[20] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, NY, USA, 2006. ACM.

[21] S. Hunt and D. Sands. Just Forget it – The Semantics and Enforcement of Information Erasure. In *Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008*, LNCS. Springer Verlag, 2008.

[22] N. Kobayashi and K. Shirane. Type-Based Information Analysis for Low-Level Languages. In *APLAS*, pages 302–316, 2002.

[23] X. Leroy. Bytecode verification on Java smart cards. *Software: Practice and Experience*, 32(4):319–340, Apr. 2002.

[24] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In Nicola [27], pages 141–156.

[25] H. Mantel and A. Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-threaded Programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[26] H. Mantel and D. Sands. Controlled Declassification Based on Intransitive Noninterference. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2004.

[27] R. D. Nicola, editor. *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007,*

Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[29] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[30] J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical report, SRI, dec 1992.

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

[32] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–273. Springer-Verlag, July 2003.

[33] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 376–394, Madrid, Spain, Sept. 2002. Springer-Verlag.

[34] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *ISSS*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.

[35] A. Sabelfeld and D. Sands. Dimensions and Principles of Declassification. In aa [1], pages 255–269.

[36] D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

[37] S. Zdancewic. Challenges for Information-flow Security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, August 2004.

Cryptographically-Masked Flows

Aslan Askarov Daniel Hedin Andrei Sabelfeld

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract. Cryptographic operations are essential for many security-critical systems. Reasoning about information flow in such systems is challenging because typical (noninterference-based) information-flow definitions allow no flow from secret to public data. Unfortunately, this implies that programs with encryption are ruled out because encrypted output depends on secret inputs: the plaintext and the key. However, it is desirable to allow flows arising from encryption with secret keys provided that the underlying cryptographic algorithm is strong enough. In this paper we conservatively extend the noninterference definition to allow safe encryption, decryption, and key generation. To illustrate the usefulness of this approach, we propose (and implement) a type system that guarantees noninterference for a small imperative language with primitive cryptographic operations. The type system prevents dangerous program behavior (e.g., giving away a secret key or confusing keys and non-keys), which we exemplify with secure implementations of cryptographic protocols. Because the model is based on a standard noninterference property, it allows us to develop some natural extensions. In particular, we consider public-key cryptography and integrity, which accommodate reasoning about primitives that are vulnerable to chosen-ciphertext attacks.

1 Introduction

Cryptographic operations are ubiquitous in security-critical systems. Reasoning about information flow in such systems is challenging because typical information-flow definitions allow no flow from secret to public data. The latter requirement underlies *noninterference* [11, 16], which demands that public outputs are unchanged as secret inputs are varied. While traditional noninterference breaks in the presence of cryptographic operations, the challenge is to distinguish between breaking noninterference because of legitimate use of sufficiently strong encryption and breaking noninterference due to an unintended leak.

A common approach to handling cryptographic primitives in information-flow aware systems is by allowing *declassification* of encryption results. The intention of declassification is that the result of encryption can be released to the attacker. Declassification, however, is a versatile mechanism: different declassification dimensions correspond to different reasons why information is released [29, 4]. Attempts at framing cryptographically-masked flows into different dimensions have been made although, as we discuss, not always with satisfactory results.

In this paper, we introduce cryptographic primitives into an information-flow setting while preserving a form of noninterference property. This is achieved by building

In *Proc. 13th International Static Analysis Symposium*, Seoul, Korea, August 2006. LNCS.

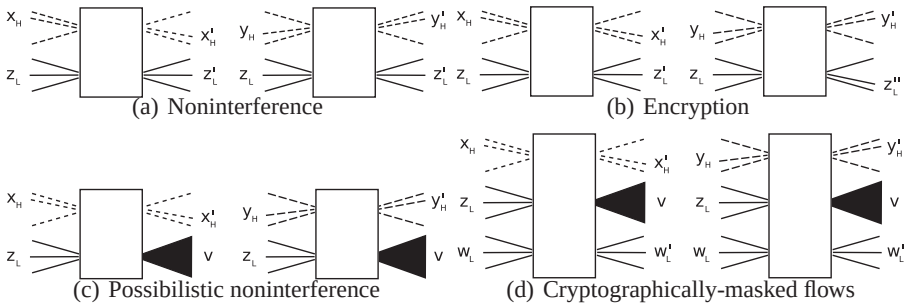

Fig. 1. From noninterference to cryptographically-masked flows

in the model a basic assumption that attackers may not distinguish between ciphertexts and that decryption using the wrong key fails. Although this assumption is stronger than some probabilistic and computational cryptographic models (which allow some information to leak when comparing ciphertexts), we argue that it can still be reasonable, and that it opens up possibilities for tracking information flow in the presence of cryptographic primitives in expressive programming languages.

The intuition behind our approach is sketched below and illustrated in Figure 1, where dashed and solid lines correspond to secret and public values, respectively. Fixing some public (low) input x_L and varying secret (high) input from x_H to y_H may not reflect on a public output z'_L of a system that satisfies noninterference (illustrated in Figure 1(a)). Suppose the system in question involves encryption, such as in the program $z = \texttt{enc}(k, x)$ for some secret key k. Clearly, noninterference is broken: variation in the secret input from x_H to y_H may cause variation in the public output from z'_L to z''_L (illustrated in Figure 1(b)).

However, noninterference can be recovered if the result of encryption is possibly *any* value v. This means that variation of the high input from x_H to y_H does not affect the public output—any value v is a possible public output in both cases. This form of noninterference is known as *possibilistic noninterference* [24] (illustrated in Figure 1(c)). Overall, although low outputs might depend on low inputs and ciphertexts, no observation about possible low outputs may reveal information about changes in high inputs (illustrated in Figure 1(d)).

This paper makes a case for possibilistic noninterference as a natural model for cryptographically-masked flows. Further, we have designed and implemented a security type system that provably enforces possibilistic noninterference for an imperative language with primitive cryptographic operations and communication channels. The type system prevents dangerous program behavior (e.g., giving away a secret key or confusing keys or non-keys), which we exemplify with secure implementations of cryptographic protocols. Because the model is based on a standard noninterference property, it allows us to develop some natural extensions. In particular, we consider public-key cryptography and integrity, which accommodates reasoning about primitives that are vulnerable to chosen-ciphertext attacks.

sec. levels	$\sigma ::= \texttt{L} \mid \texttt{H}$	basic types	$t ::= \texttt{int} \mid \texttt{enc}_\gamma\ \tau$
key levels	$\gamma ::= \texttt{P} \mid \texttt{S}$	prim. types	$\tau ::= t\ \sigma \mid \texttt{key}\ \gamma \mid (\tau_1, \tau_2)$
global decls.	$gd ::= \texttt{global}\ x\ \gamma \mid ch\ \tau$	local decls.	$ld ::= x\ \tau$

expressions $e ::= n \mid x \mid e_1\ op\ e_2 \mid \texttt{enc}_\gamma\ (e_1, e_2) \mid \texttt{dec}_\gamma\ (e_1, e_2) \mid \texttt{newkey}\ \gamma \mid (e_1, e_2)$
$\mid\ \texttt{fst}(e) \mid \texttt{snd}(e)$

statements $c ::= \texttt{skip} \mid x := e \mid \texttt{if}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2 \mid \texttt{while}\ e\ \texttt{do}\ b \mid \texttt{out}(ch, e)$
$\mid\ \texttt{in}(x, ch)$

block $b ::= \{ld_1; \ldots ld_n; c_1; \ldots; c_m\}$

actor $actor ::= A\ b$ program $prog ::= gd_1; \ldots gd_n; actor_1 \ldots actor_m$

Fig. 2. Syntax

2 Language

We explore how to model cryptographic flows in a small imperative language equipped with primitive encryption functions, dynamic key generation, and channels for communication. This section introduces the syntax and semantics of the language. For space reasons we are forced to omit the standard features of the language. The complete rules can be, however, found in the full version of this paper [3].

Syntax The syntax of the language is defined in Figure 2. Let $x \in VarName$ range over the set of variable names and $ch \in ChanName$ range over the set of channel names. A *program* consists of a sequence of *global declarations* followed by a sequence of *actors*. A global declaration is either a declaration of a *global key* or the declaration of a channel. Global keys are declared by associating a variable name with a *key level*. Values and keys have corresponding *security levels*. Values are either *public (low)* `L` or *secret (high)* `H`. The key levels declare the maximum value security level the key can safely encrypt. In particular, a key of level `S` may safely encrypt public and secret values, whereas a key of level `P` may only safely encrypt public values. Let $KeyLvl = \{\texttt{S}, \texttt{P}\}$ be the set of key levels. Global keys are assumed to have appropriate values at the beginning of the execution of a program and correspond to initial shared secrets between the actors of the program. A *channel* is declared by associating a channel name with the type of the messages that will be sent over the channel. Let A range over the set of *actor names*. An actor is defined by naming a *block*, representing the code of the actor. A block is simply a sequence of variable declarations followed by a sequence of commands. Variables are local to the block in which they are declared. The commands include the standard commands of an imperative language and commands for sending on and receiving from a given channel. Apart from expressions for generating new keys and for encryption and decryption, expressions are standard: integers, variables, total binary operators, pair formation, and projection.

Semantics The semantics of the system is defined as a big-step operational semantics. The actors of a program run concurrently and interact with each other by sending and receiving messages on the declared channels. We refrain from modeling the semantics for the entire system and instead provide semantics for isolated actors. Thus we deliberately ignore information flows via races and other flows that may arise in concurrent

systems (cf. [27]). First we define the values and environments, which are used in the following definition of the semantics of expressions and commands. Let $n \in \mathbb{Z}$ range over the *integers* and $k \in Key = Key_{\mathtt{P}} \cup Key_{\mathtt{S}}$ range over *keys*, where $Key_{\mathtt{P}}$ and $Key_{\mathtt{S}}$ are disjoint. The *values* are built up by the *ordinary values*, integers, keys and *pairs* of values, together with the *encrypted values* $u \in U = U_{\mathtt{P}} \cup U_{\mathtt{S}}$.

$$\text{values} \in \mathit{Value} \qquad v ::= n \mid k \mid (v_1, v_2) \mid u$$

The system is parameterized over two *symmetric encryption schemes*—one for each key level γ—represented by triples $S\mathcal{E}_\gamma = (\mathcal{K}_\gamma, \mathcal{E}_\gamma, \mathcal{D}_\gamma)$, where

- $\mathcal{K}_\gamma$ is a *key generation* algorithm that on each invocation generates a new key.
- $\mathcal{E}_\gamma$ is a *probabilistic* encryption algorithm that takes a key $k \in Key_\gamma$, a value $v \in \mathit{Value}$ and returns a ciphertext $u \in U_\gamma$.
- $\mathcal{D}_\gamma$ is a deterministic decryption algorithm that takes a key $k \in Key_\gamma$, a ciphertext $u \in U_\gamma$ and returns a value $v \in \mathit{Value}$ or fails. Decryption should satisfy $\mathcal{D}_\gamma(k, \mathcal{E}_\gamma(k, v)) = v$.

The reason for the use of different encryption schemes for different security levels is to lay the ground for an extension of the system into a *multi-level* system, i.e. a system with more than two security levels. In such a system we would have one encryption schema at each security level, trusted to encrypt values up to and including the security level. We shall assume that the keys sets $Key_{\mathtt{P}}$ and $Key_{\mathtt{S}}$ of the two different encryption schemes are distinct; let pk range over $Key_{\mathtt{P}}$ and sk over $Key_{\mathtt{S}}$.

Input and output is modeled in terms of streams of values with the cons operation "$\cdot$" and the distinguished empty stream ϵ. The *full environment* E consists of four components: (i) the variable environment M, which is a stack of mappings from variable names to lifted values (values joined with a special value for undefined $\mathit{Value}^\bullet = \mathit{Value} \cup \{\bullet\}$); (ii) the key-stream environment G, which maps an encryption scheme level to the *stream of keys* generated by successive use of the key generator (let ks range over streams of keys); (iii) the input environment I and (iv) the output environment O, which map channel names to streams of values.

Semantics of Expressions The evaluation of expressions has the form $\langle (M, G), e\rangle \Downarrow \langle G', v\rangle$: evaluating an expression in a given variable and key-stream environment yields a value and a possibly updated key-stream environment. The semantics of integers, variables, total binary operators, pair formation, and projection are entirely standard.

Figure 3 presents the rules specific to the treatment of cryptography; the rest of the rules can be found in [3]. Key generation (S-NEWKEY) takes the level of the key to be generated and returns the topmost element in the key stream associated to that level in the key-stream environment. Encryption (S-ENC) and decryption (S-DEC) both use the encryption schemes $S\mathcal{E}_\gamma$ introduced above.

Semantics of Commands Commands are state transformers of the form $\langle E, c\rangle \Downarrow E'$: the command c yields the new environment E' when run in the environment E. The semantics of the commands is entirely standard for a while language with channels—everything specific to encryption is in the expressions. For space reasons the semantics of the commands is not presented here but can be found in [3].

$$\text{(S-NEWKEY)}\ \frac{G(\gamma) = k \cdot ks}{\langle (M, G), \texttt{newkey}\ \gamma\rangle \Downarrow \langle G[\gamma \mapsto ks], k\rangle}$$

$$\text{(S-ENC)}\ \frac{\langle (M, G), e_1\rangle \Downarrow \langle G', k\rangle \quad \langle (M, G'), e_2\rangle \Downarrow \langle G'', v\rangle \quad k \in Key_\gamma \quad u = \mathcal{E}_\gamma(k, v)}{\langle (M, G), \texttt{enc}_\gamma\ (e_1, e_2)\rangle \Downarrow \langle G'', u\rangle}$$

$$\text{(S-DEC)}\ \frac{\langle (M, G), e_1\rangle \Downarrow \langle G', k\rangle \quad \langle (M, G'), e_2\rangle \Downarrow \langle G'', u\rangle \quad k \in Key_\gamma \quad v = \mathcal{D}_\gamma(k, u)}{\langle (M, G), \texttt{dec}_\gamma\ (e_1, e_2)\rangle \Downarrow \langle G'', v\rangle}$$

Fig. 3. Semantics of Expressions

3 Security

This section states the assumptions our semantic model makes on the underlying encryption schema and shows how these assumptions lead up to a natural formulation of possibilistic noninterference. The section concludes by investigating the relation between our assumptions and common cryptographic attacker models.

Encryption Model As was mentioned above, this paper only considers *probabilistic encryption schemes*. A probabilistic encryption scheme is a triple $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ where the encryption algorithm is a function from a key, a plaintext, and some initial *random data*, referred to as the *initial vector*. Such an algorithm will produce a set of possible ciphertexts for each plaintext-key pair, one ciphertext for each initial vector.

To be able to formulate and prove possibilistic noninterference for our system we need to demand two properties of the underlying encryption schemes. The first property is the assumption that an adversary can learn nothing about the plaintext or the key by observing the ciphertext. This property, known as Shannon's *perfect secrecy* [30], is used to justify our *indistinguishability* relation on ciphertexts.

The second property is an *authenticity property* needed in the treatment of decryption. More precisely we are assuming that decryption using the *wrong* key fails:

$$\mathcal{D}(k, \mathcal{E}(k', v)) = \bot \text{ if } k \neq k'$$

Insufficiency of Standard Noninterference The prevailing notion when defining confidentiality in the analysis of information flows is noninterference. Noninterference is typically formalized as the preservation of a *low-equivalence* relation under the execution of a program: if a program is run in two low-equivalent environments then the resulting environments should be low-equivalent. For ordinary values like integers low-equivalence demands that public values are equal. However, from the assumption that an adversary can learn nothing about the plaintext from observing the ciphertext it is secure to treat all ciphertexts of the same length[1] as low-equivalent. However appealing this may be, such a treatment leads the ability of masking implicit flows in ciphertexts. Consider the program on Listing 1 for some public channel ch and encryption with secret key k:

[1] We do not assume that encryption hides the length of messages.

If all encrypted values are considered equal then we cannot distinguish between the first and the second output value, even though it is clear that the equality/inequality of the first and the second value reflects the secret value h.

```
l := enc(k, a);
out(ch, l);
if (h) then l := enc(k, b) else skip;
out(ch, l);
```

Listing 1. Occlusion

Possibilistic Noninterference To address this problem we use a variant of noninterference known as possibilistic noninterference, which allows us to create a notion of low-equivalence that disallows the above example without disallowing intuitively secure uses. Before we formalize our notion of possibilistic noninterference, let us lift the evaluation relation to a set of results as follows:

$$\langle E, c\rangle \Downarrow \hat{E} \text{ iff } \hat{E} = \{E' \mid \langle E, c\rangle \Downarrow E'\}$$

With this we can formulate our notion of possibilistic noninterference. Let $E_1 \sim_\Sigma E_2$ denote that the environments E_1 and E_2 are low-equivalent w.r.t the environment type Σ. A pair of commands, c_1 and c_2 are noninterfering if

$$\begin{aligned} NI(c_1, c_2)_\Sigma \equiv \forall E_1, E_2 \,.\, E_1 \sim_\Sigma E_2 \wedge \\ \langle E_1, c_1\rangle \Downarrow \hat{E}_1 \wedge \hat{E}_1 \neq \emptyset \wedge \langle E_2, c_2\rangle \Downarrow \hat{E}_2 \wedge \hat{E}_2 \neq \emptyset \Longrightarrow \\ \forall E_1' \in \hat{E}_1 \exists E_2' \in \hat{E}_2 \,.\, E_1' \sim_\Sigma E_2' \end{aligned}$$

That is, two commands are considered *equivalent* if, for every pair of low-equivalent environments *in which the commands terminate* it holds that there exists the *possibility* that each environment produced by the first command when run in the first environment can be produced by the second command when run in the second environment.

By only considering environments for which the commands terminate, we ignore the issue with crashes. This is equivalent to saying that normal and abnormal termination cannot be distinguished by the attacker.

Adequacy of the Model The choice of possibilistic noninterference does not automatically solve the above problem—using the full low-equivalence relation on ciphertexts would lead to the same danger of masking insecure flows. Instead the low-equivalence relation has to be crafted carefully to avoid masked insecure flows and at the same time allow secure usage of encryption primitives. We will now show how this can be done for probabilistic encryption schemes. Consider first what happens in the above example. Let two low-equivalent environments E_1 and E_2 s.t. h is *true* in the first and *false* in the second. The result of running the *if* statement of the example above in the second environment E_2 is the singleton set $\hat{E}_2 = \{E_2\}$. However, the result of running it in the first environment is the set of environments $\hat{E}_1 = \{E_1[l = c] \mid encrypt(b) = c\}$, where each c is obtained by encrypting b under the same key but with different initial vectors. The demand of possibilistic noninterference is that for each environment in $\hat{E}_1$ there should exists a low-equivalent environment in $\hat{E}_2$. This is only the case if all ciphertexts $\{c \mid encrypt(b) = c\}$ are low-equivalent. Thus, any low-equivalence relation that does not consider the different ciphertexts originating from one plaintext and one

key to be the equivalent will prevent this kind of masking. However, we must make sure that each ciphertext produced by one plaintext and key has a low-equivalent ciphertext for each other choice of plaintext and key.

Fortunately, for probabilistic encryption schemes we can easily form a low-equivalence relation $\doteq$ with these properties by regarding ciphertexts *with the same random initial vector* to be equivalent:

$$\forall k_1, k_2, v_1, v_2 \,.\, \mathcal{E}(k_1, v_1, iv) \doteq \mathcal{E}(k_2, v_2, iv)$$

where iv ranges over initial vectors. This relation has the following properties: (i) different ciphertexts produced by one plaintext and one key will have different initial vectors and will not be low-equivalent, and (ii) since each plaintext and key will produce ciphertexts using all initial vectors, for each ciphertext produced by one plaintext and key there will be exactly one low-equivalent ciphertext for every other choice of plaintext and key.

Relation to Computational Adversary Models The perfect secrecy and authenticity demands on the encryption schemes are fairly strong. However, there are schemes for which the probability of breaking these assumptions is provably negligible.

The first demand that the ciphertexts should give no information about the plaintexts is commonly relaxed to the notion of *semantic security under chosen plaintext attack* (SEM-CPA) by assuming that the adversary has *limited computational power*. Semantic security states that "*Whatever is efficiently computable about the cleartext given the cyphertext, is also efficiently computable without the cyphertext*" [17]. [2]

In the same way we may allow a relaxation of the demand of authenticity, which can be implemented by combining *Message Authentication Code* (MAC) with a SEM-CPA encryption scheme to form a new scheme that is both secure (SEM-CPA) and authenticity preserving (INT-PTXT)[6]. A scheme is INT-PTXT if the chance that an adversary can produce ciphertexts C s.t. $M = \mathcal{D}_k(C) \neq \perp$ and M was never a parameter of $\mathcal{E}_k(\cdot)$ is negligible. To see that the probability of a successful decryption using the wrong key is negligible under an INT-PTXT scheme consider the following. If a ciphertext $C = \mathcal{E}_k(M)$ decrypts successfully using another key than was used to construct the message i.e. $M' = \mathcal{D}_{k'}(C)$ for $k' \neq k$ then the scheme cannot be INT-PTXT, since M' was never a parameter of $\mathcal{E}_{k'}(\cdot)$.

On Semantic Security We believe that it is possible to prove a general result that if a program with SEM-CPA + INT-PTXT encryption primitives is secure w.r.t. possibilistic noninterference then it is also semantically secure. This result is likely to involve restrictions on *key cycles*, which are a known problem when reconciling the formal and computational views of cryptography [2], or demanding that the underlying schema is secure in the presence of such cycles (cf. *KDM security* [7]).

With such a result at hand, we shall be able to capitalize on the modularity of our approach. For a given language and type system, as soon as we can prove that all well-typed programs are noninterfering, we automatically get semantic security. This opens

[2] There is another frequently used notion of security under a computationally limited adversary, IND-CPA. IND-CPA has been shown to be equivalent to SEM-CPA [17, 6].

up possibilities for reasoning about expressive languages and type systems, where all we have to worry about are noninterference proofs (which are typically simpler than proofs of computational soundness).

4 Types

The syntax of the types is defined in Figure 2. A *primitive type* is either a *security annotated basic type*, a pair of primitive types or a *key type*. The security annotation assigns a security level to the basic type expressing whether it is *secret* or *public*. The types of encrypted values are *structural* in the sense that the type reflects the original type of the encrypted values as well as the level of the key that was used in the encryption. For instance, $\texttt{enc}_\texttt{S}\ (\texttt{int H})\ \texttt{L}$ is the type of a secret integer that has been encrypted with a secret key once and $\texttt{enc}_\texttt{S}\ (\texttt{enc}_\texttt{S}\ (\texttt{int H})\ \texttt{L})\ \texttt{L}$ is the type of an integer that has been encrypted with a secret key twice. The type of the variable environment Ω is a map from variables to primitive types, the type of the input environment and the output environment alike Θ is a map from channel names to primitive types, and the key-stream environment defines its own type (in the domain of the environment). The type of the entire environment, Σ, is the pair of a variable type environment and a channel type environment.

Well-formed Values Well-formedness defines the meaning of the types ignoring the security annotations. The well-formedness is entirely standard and is omitted for space reasons.

Low-equivalence In Figure 4 we formalize the low-equivalence relation. For complex types, i.e., pairs and environments, low-equivalence is defined structurally by demanding the parts of the complex type to be low-equivalent w.r.t. the corresponding type. Any values are low-equivalent w.r.t. a secret type. Integers are low-equivalent w.r.t. a public integer type if they are equal. Low-equivalence for keys is slightly different since keys are not annotated with a security level—only a key level—whose meaning is defined by well-formed values as different sets. Even though it is semantically meaningful to add a security level to key types—the values of keys can be indirectly affected by computation—we have chosen not to. Instead, a public key is considered to be of low security and a secret key of high security. Thus, public keys are low-equivalent if they are equal, and any two secret keys are low-equivalent.

The most interesting rule is the rule defining low-equivalence w.r.t. a public encryption type (LE-ENC-L1) and (LE-ENC-L2). These two rules define the difference in meaning between encryption with a secret and a public key. First, in both rules, the encrypted values must be low-equivalent w.r.t. the low-equivalence relation of encrypted values. Second, there must exist a pair of low-equivalent keys w.r.t. the key type of the encryption type that decrypt the encrypted value to two values. This is where the rules differ. Since ciphertexts created by public keys can be decrypted by anyone with access to the public keys, we have to demand that the inside of the encrypted value contains only public values. This is done in the (LE-ENC-L2) rule, which demands that the inside

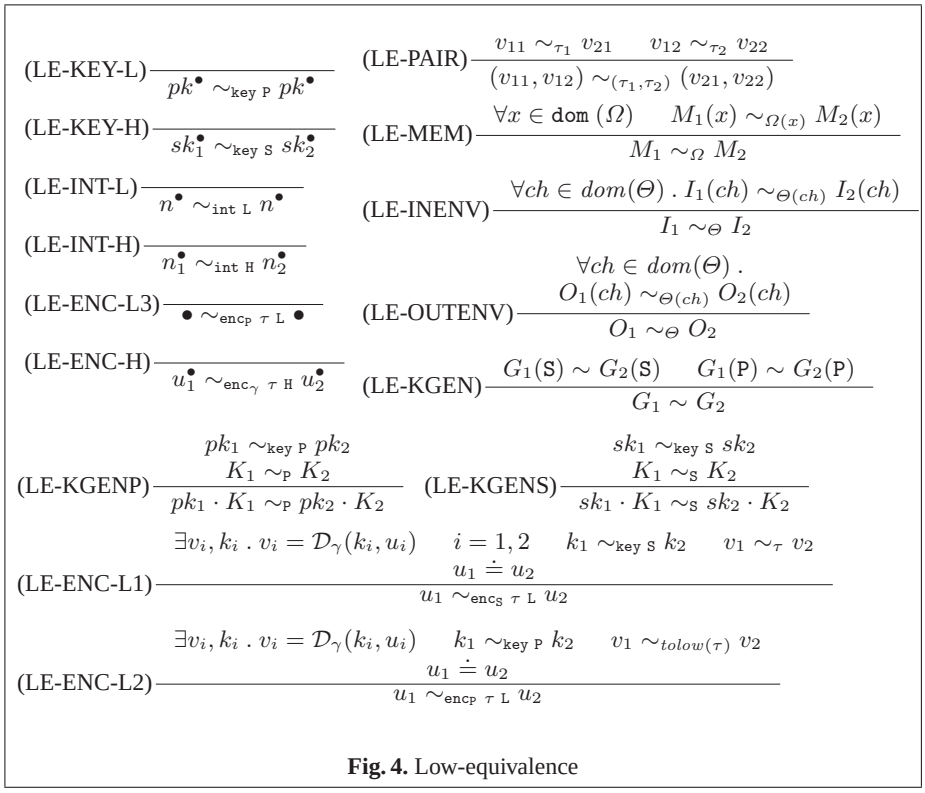

Fig. 4. Low-equivalence

is not only low-equivalent w.r.t. its type τ, but low-equivalent w.r.t. $tolow(\tau)$, which is defined as follows:

$$tolow(t\ \sigma) = t\ \mathtt{L} \quad tolow(\mathtt{key\ P}) = \mathtt{key\ P} \quad tolow((\tau_1, \tau_2)) = (tolow(\tau_1), tolow(\tau_2))$$

The (LE-ENC-L1) rule can be seen as encoding the power of the attackers. For encryption with secret keys the demand is only that the resulting values should be low-equivalent w.r.t. the primitive type, τ, of the encryption type. This way, we demand low-equivalence inside encrypted values and make certain that that the result of decrypting low-equivalent encrypted values will result in low-equivalent values and that secret values are not stored inside encrypted values that are created by public keys.

Subtyping The subtyping is entirely standard; it allows public information to be seen as secret with the exception of invariant subtyping for keys. The subtyping relation for primitive types, $<:$, and the subtyping relation for security levels, $\sqsubseteq$, defines the corresponding join operators. The subtyping relation can be found in [3].

Expression Type Rules The type rules for expressions are of the form $\Omega, pc \vdash e : \tau$. Figure 5 defines typing rules for non-standard expressions, while the rest of the rules can be found in [3]. The generation of a new key with the requested security level results in a key with that security level if the requested level is not below the context type. The

$$\text{(T-NEWKEY)}\frac{pc \sqsubseteq lvl(\texttt{key}\ \gamma)}{\Omega, pc \vdash \texttt{newkey}\ \gamma : \texttt{key}\ \gamma} \quad \text{(T-ENC1)}\frac{\Omega, pc \vdash e_1 : \texttt{key S} \quad \Omega, pc \vdash e_2 : \tau}{\Omega, pc \vdash \texttt{enc}_\texttt{S}\ (e_1, e_2) : \texttt{enc}_\texttt{S}\ \tau\ \texttt{L}}$$

$$\text{(T-ENC2)}\frac{\Omega, pc \vdash e_1 : \texttt{key P} \quad \Omega, pc \vdash e_2 : \tau \quad lvl(\tau) = \sigma}{\Omega, pc \vdash \texttt{enc}_\texttt{P}\ (e_1, e_2) : \texttt{enc}_\texttt{P}\ \tau\ \sigma} \quad \text{(T-DEC)}\frac{\Omega, pc \vdash e_1 : \texttt{key}\ \gamma \quad \Omega, pc \vdash e_2 : \texttt{enc}_\gamma\ \tau\ \sigma}{\Omega, pc \vdash \texttt{dec}_\gamma\ (e_1, e_2) : \tau^\sigma}$$

Fig. 5. Type Rules of Expressions

reason for this is that we assume that the public-key stream is publicly observable. Encryption with secret keys will always result in public encrypted values. Encryption with public keys is possible on any value but produces a result that is as secret as the original value. Both the type rule for key generation and the type rule for public encryption makes use of function $lvl(\cdot)$ that computes the security level of the given value:

$$lvl(t\ \sigma) = \sigma \quad lvl((\tau_1, \tau_2)) = lvl(\tau_1) \sqcup lvl(\tau_2) \quad lvl(\texttt{key P}) = \texttt{L} \quad lvl(\texttt{key S}) = \texttt{H}$$

Decryption is allowed only if the key level of the key used for decryption matches the key level of the encrypted value. The result of the decryption is tainted by the security level of the encrypted values. The taint function is defined as follows:

$$(t\ \sigma)^{\sigma'} = t\ (\sigma \sqcup \sigma') \quad (\tau_1, \tau_2)^\sigma = (\tau_1^\sigma, \tau_2^\sigma) \quad (\texttt{key P})^\texttt{L} = \texttt{key P} \quad (\texttt{key S})^\sigma = \texttt{key S}$$

Command Type Rules As with expressions most of the rules are standard for a security type system (cf. [34]). As is standard, following Denning's original approach to analyzing programs for secure information flow [13], in order to prevent implicit flows the notion of *security context* is defined. The security context of a program point is defined to be the least upper bound of the security levels of the conditional expressions of the enclosing conditionals. The context affects the the commands with side-effects, i.e., variable assignment, input, and output. A block of local declarations followed by a sequence of statements is checked by first adding the declared variables to the variable environment and then checking all statements in the new type environment. The type rule for sequences of statements (T-SEQ) checks all statements of the sequence. *If* and *while* are the two constructs that can lead to indirect flows since they affect the control flow. Thus, the body of the *if* and the *while* are checked in the context of the security level of the control expression. This way, when a branch is depending on a secret the body of that branch is prevented from causing any low side effects. The type rules of commands can be found in [3].

5 Soundness

The main soundness theorem of the paper states that well-typed programs are noninterfering. Typically, for typed programming languages, the soundness is phrased in terms

of *progress*, i.e. well-typed programs can always be evaluated in well-formed environments, and *preservation*, i.e. after this step has been made the resulting environment is well formed. It may be interesting to note that the way we have avoided to model error makes this system not satisfy progress: decryption with the wrong key or computing with an uninitialized variable will prevent evaluation. The well known solution is to model failure in the semantics. To keep the presentation cleaner we refrain from this.

The soundness theorem states that well-typed programs are noninterfering. Section 3 lifts the evaluation relation of commands to sets and formulates noninterference for commands. Before giving the formulation of the soundness theorem we must lift the codomain of the evaluation relation of expressions to sets and formulate noninterference for expressions:

$$\langle (M, G), e\rangle \Downarrow \langle G', \hat{v}\rangle \text{ iff } \hat{v} = \{v \mid \langle (M, G), e\rangle \Downarrow \langle G', v\rangle\}$$

With this we can define noninterference for expressions, which is equivalent to the noninterference of statements defined above. Put simply, if two expressions e_1 and e_2 are run in low-equivalent key-stream and variable environments, yielding pairs of new key-stream environments and results, then these results should be low-equivalent:

$$\begin{aligned} NI(e_1, e_2)_{\Omega,\tau} \equiv \forall M_1, M_2, G_1, G_2 \,.\, M_1 \sim_\Omega M_2 \wedge G_1 \sim G_2 \wedge \\ \langle (M_i, G_i), e_i\rangle \Downarrow \langle G_i', \hat{v}_i\rangle \wedge \hat{v}_i \neq \emptyset \Longrightarrow \\ G_1' \sim G_2' \wedge \forall v_1 \in \hat{v}_1 \; \exists v_2 \in \hat{v}_2 \,.\, v_1 \sim_\tau v_2 \end{aligned}$$

We arrive at the soundness theorems for expressions and commands, both proved by induction on type derivation [3].

Theorem 1. *Soundness for expressions* $\Omega, pc \vdash e : \tau \Longrightarrow NI(e, e)_{\Omega,\tau}$

Theorem 2. *Soundness for commands* $\Sigma, pc \vdash c \Longrightarrow NI(c, c)_\Sigma$

6 Extensions

In this section we consider two extensions: integrity and public-key cryptography.

Integrity Confidentiality classifies information into public and secret, i.e., information that may or may not be given to the world, respectively. Dually, integrity classifies information into *untrusted* (or *low-integrity*) and *trusted* (or *high-integrity*), i.e., whether the information may or may not have been *affected* by the world.

Tracking the integrity of data enables us to explore some additional dimensions of cryptography: weaknesses of the encryption algorithms and the effect of encryption on integrity. Consider for example, a primitive that is vulnerable to chosen ciphertext attacks. With integrity controls, it is natural to express the restriction that untrusted encrypted values may not be decrypted.

In the presence of integrity the security levels for values are pairs of the form (σ, ι), where σ is a confidentiality level, and ι is a corresponding integrity level. The following tables define two functions—$\texttt{safe}_{\mathcal{E}}(\alpha, (\sigma, \iota))$ and $\texttt{safe}_{\mathcal{D}}(\alpha, (\sigma, \iota))$—that indicate

if it is safe to encrypt (decrypt) a plaintext (ciphertext) of security level (σ, ι) with an encryption scheme that has property α. Here α ranges over standard notions [5]—IND-CCA (indistinguishable under chosen-ciphertext attacks) and IND-CPA (indistinguishable under chosen-plaintext attacks).

	(H,H)	(L,L)	(H,L)	(L,H)
IND-CCA	safe	safe	safe	safe
IND-CPA	safe	safe	safe	safe

$\texttt{safe}_{\mathcal{E}}(\alpha, (\sigma, \iota))$

	(H,H)	(L,L)	(H,L)	(L,H)
IND-CCA	safe	safe	safe	safe
IND-CPA	safe	-	-	safe

$\texttt{safe}_{\mathcal{D}}(\alpha, (\sigma, \iota))$

In this way we can provide different type rules for different assumptions on the vulnerability properties of the encryption and decryption algorithms:

$$\text{(T-ENC*)}\ \frac{\Omega, pc \vdash e_1 : \texttt{key S} \qquad \Omega, pc \vdash e_2 : \tau \qquad lvl(\tau) = (\sigma, \iota) \qquad \texttt{safe}_{\mathcal{E}}(\alpha, (\sigma, \iota))}{\Omega, pc \vdash \texttt{enc}_{\texttt{S}}^{\alpha}\ (e_1, e_2) : \texttt{enc}_{\texttt{S}}\ \tau\ (\texttt{L}, \texttt{H})}$$

$$\text{(T-DEC*)}\ \frac{\Omega, pc \vdash e_1 : \texttt{key}\ \gamma \qquad \texttt{safe}_{\mathcal{D}}(\alpha, (\sigma, \iota)) \qquad \Omega, pc \vdash e_2 : \texttt{enc}_{\gamma}\ \tau\ (\sigma, \iota)}{\Omega, pc \vdash \texttt{dec}_{\gamma}^{\alpha}\ (e_1, e_2) : \tau^{(\sigma, \iota)}}$$

A Note on the Integrity of Keys The current model allows very limited interaction with keys apart from encryption. Since the values of keys cannot be programmatically inspected, the power of the attacker is limited to choice between secure keys. Thus, the model cannot in its present form distinguish between encryption with high and low-integrity keys w.r.t. *confidentiality*. The intuition is clear: since the attacker can only choose between secure keys, that choice will give different but safe encrypted values.

Public-Key Cryptography Even though the present system deals only with symmetric-key cryptography, there is nothing in the model that prevents modeling public-key cryptography. The set of secret keys would contain the *private* keys and the set of public keys would contain the *public* keys, where the private keys and the public keys are dual. In this system values encrypted with public keys would be considered public, since only actors with access to the private keys would be able to decrypt them.

However, public-key cryptography is most interesting in the presence of integrity. In the same way we can model that encryption of secrets using secret keys results in public values, we can model that encryption raises the integrity of the encrypted value to the integrity of the key, which corresponds to signing.

7 Programming with encryption: Examples

We have implemented a prototype of the type system and mechanically type-checked two applications: secure backup and a Wide-Mouthed-Frog protocol implementation. In both examples the type system prevents dangerous insecurities such as sending sensitive unencrypted data over a public channel or not using a secret key for encryption. This section discusses some interesting fragments of these implementations.

Secure Data Backup In the secure backup scenario a low-confidentiality channel is used for sending sensitive information to the remote storage. Listing 2 presents the code for the backup operation. Here and below we slightly simplify the syntax with respect to Figure 2 for the sake of readability.

```
1 global K secret;
2 backup enc secret (int high) low;
3
4 actor Backup {
5   data int high;
6   ctxt enc secret (int high) low;
7   data := ...
8   ctxt := encrypt(K, data);
9   out backup ctxt;
10 }
```

Listing 2. Backup code

Here, the global declarations contain secret key `K` and low channel `backup`. The type of the latter says that only encrypted high integers may be sent over this channel.

Lines 5 and 7 declare and initialize a high integer variable `data`. Line 6 declares the variable `ctxt` of type `enc secret (int high) low`. On line 8 the value of variable `data` is encrypted with secret key `K` and the resulting ciphertext is assigned to the variable `ctxt`. Since type of `ctxt` matches the type of the `backup` channel it might be sent over this channel. This is done by the `out` command on line 9.

```
1 actor Restore {
2   data int high;
3   ctxt enc secret (int high) low;
4   in ctxt backup;
5   data := decrypt(K, ctxt);
6 }
```

Listing 3. Recovery code

When recovering data, an actor reads the data from the public channel and decrypts it. Assuming the same global declarations Listing 3 presents the recovery code. Here, line 4 reads data from the `backup` channel. It's decrypted using the key `K` on line 5.

An example of an easy-to-overlook error is to have the following line in place of line 9 in the body of actor `Backup`: `out backup data;`. This is an insecurity that the type system rejects. Generally, in the secure backup example the type system ensures that secret data is encrypted before it is sent over the `backup` channel, thus preventing accidental leaks.

Wide-Mouthed-Frog Protocol The Wide-Mouthed-Frog protocol [8] is a simple key exchange protocol with trusted server and timestamps. In this protocol secret keys K_{AS} and K_{BS} are shared between server S and principals A and B, respectively. Principal A generates a fresh session key K_{AB}, which is transferred to B in two messages:

$$1.\ A \to S : A, \{T_A, B, K_{AB}\}K_{AS}$$
$$2.\ S \to B : \{T_S, A, K_{AB}\}K_{BS}$$

The first message consist of A's name and a tuple encrypted with the shared key K_{AS}. This tuple contains three elements—a timestamp T_A, the name of principal B, and a generated key K_{AB}. Upon receipt of this message, S decrypts it, checks the timestamp, replaces T_A with its own timestamp T_S, encrypts it with key K_{BS}, and forwards the resulting message to B. Principal B then checks whether the second message is timely.

Obviously, there is more to implementation of the protocol than expressed by the two-step description. Our type system guarantees that implementations do not introduce information-flow leaks in the protocol. Listing 4 presents the implementation of this protocol for principal A. The full version of this paper [3] contains the implementation for the server S and principal B.)

```
1 global Kas secret;
2 chanS <int low, enc secret
3    (<int low, <int low, key secret>>) low>;
4 chanAB enc secret (int high) low;
5 actor A {
6   idA int low; idB int low; tsA int low;
7   messageToB int high;
8   Kab key secret;
9   //  ... initialization
10  Kab := newkey (secret);
11  out chanS <idA,
12        encrypt(Kas, <tsA,<idB, Kab>>)>;
13  out chanAB encrypt (Kab, messageToB);
14 }
```

Listing 4. WMF Implementation

This program declares two channels: `chanS` for communicating with the server, and `chanAB` for sending messages to B, once the key has been exchanged. The type of the channel `chanS` corresponds to the first message in the protocol—a pair consisting of a low integer and an encryption with secret key of a three-element tuple (expressed by nested pairs). Since the level of the key used for encrypting this tuple is `secret`, it is safe to label the result of encryption as `low`. The body of the `actor` declaration defines low-confidentiality variables `idA` and `idB` that stand for the names of the principals; variable `tsA` stores the current timestamp; the high-confidentiality variable `messageToB` contains the information that A wants to send to B.

The new key is generated on line 10. Line 12 constructs the first message of the protocol and sends it to the server. Line 13 uses the newly generated key and sends the secret message to the principal B.

In this example, the type system prevents non-secret session keys in the key establishment protocol. As in the previous example, it also guarantees that secret information may not leave the system unless it is encrypted with a secret key.

8 Related work

As mentioned in the introduction, declassification models are sometimes used to justify cryptographic primitives in languages with information-flow control. Declassification mechanisms facilitate information release. A recent classification of declassification [29] suggests that information release policies represent aspects of *what* is declassified, by *whom*, *when* and *where* in the system. These correspond to dimensions of information release. The relation of our model to declassification is somewhat subtle, because masking does not actually model information release. Hence, none of the release dimensions is directly suitable for cryptographically-masked flows.

Furthermore, attempts at framing cryptographically-masked flows into different dimensions do not always lead to satisfactory results. For example, releasing the difference between two values of a secret whenever the results of its encryption are different can be a deceptive policy when assumptions about the underlying cryptographic primitives are not explicitly stated. If the underlying encryption function is bijective (assuming the key is fixed) then releasing the result of encryption is equivalent to releasing the secret itself. This phenomenon applies to typical policies from the *what* dimension, such as delimited release [28].

Another example of releasing the secret itself, together with the result of a cryptographic primitive applied to the secret, can be found in [9]. The password checker example is based on matching the hash of the password with the hash of a user query. The password has a label $H \overset{cert}{\rightsquigarrow} L$, which means that the level of the password is even-

tually declassified from high to low. This, however, allows the password itself to be released to the attacker in cleartext.

Nevertheless, declassification is meaningful in the context of cryptographic computation when the attacker is capable of learning some information from ciphertext. Temporal policies express *when*, at earliest, the attacker might learn the secret. Volpano and Smith's relative secrecy [33, 32] guarantees that the attacker cannot learn the secret in polynomial time in the size of the secret. Approaches by Laud [20, 21], Laud and Vene [22], provide computational guarantees for a simple imperative language but with the assumption that keys can be statically distinguished. Mitchell et al. [23, 25] reason about security with respect to polynomial-time attackers for a form of the π calculus.

A source of our inspiration is Abadi's secrecy model for symmetric-key cryptographic protocols [1]. This model assumes that an attacker is unable to decrypt ciphertexts encrypted with secret keys. Compared to [1], we end up with simpler typing rules. For example, because of the probabilistic encryption assumption, we do not need to deal with explicit confounders. In addition, our approach accommodates natural extensions with integrity and public-key cryptography. Another source of inspiration is a logical relations technique by Sumii and Pierce that facilitates manual security proofs for cryptographic protocols [31]. This technique is not accompanied by static enforcement mechanisms (such as a type system), however.

Gordon and Jeffrey [18] extend Abadi's work to multiple security levels that may be dynamically created and may become compromised. This and other work within Gordon and Jeffrey's Cryptyc project, however, relies on trace-based properties (such as correspondence) that are weaker than noninterference. Dam and Giambiagi's work on *admissibility* [12, 15] focuses on protocol implementation, with the goal that information leaks in the implementation must adhere to those declared in protocol specification.

Duggan's and Chothia et al.'s cryptographic types [14, 10] help enforce security for a distributed programming language. This is realized through a combination of static and dynamic checks, leading to access-control guarantees (albeit without information-flow guarantees) for secrecy and integrity. Myers et al.'s qualified robustness [26] is based on a possibilistic treatment of *endorsement*, operation dual to declassification.

Hicks et al. [19] define a notion of *noninterference modulo trusted functions*, which requires parts of programs free of cryptographic functions to be in a certain sense indistinguishable. The cryptographic functions are trusted to release information if their security labels satisfy trust constraints. It is a worthwhile direction for future work to formally investigate the relation to *noninterference modulo trusted functions*. We do not expect it to be straightforward because the definition of the indistinguishability relation from [19] involves two-level semantics.

9 Conclusions and future work

We have developed an approach to tracking information flow in the presence of cryptographic operations, based on possibilistic noninterference. We have argued that a possibilistic treatment of cryptographic operations leads to a natural model of attackers that may not distinguish between ciphertexts. This model has a close connection to probabilistic encryption and, we believe, it naturally connects to computational adversary models (cf. Section 3).

Our case for possibilistic noninterference is driven by the possibility of capitalizing on the available machinery for reasoning about noninterference in programming languages. We have demonstrated that possibilistic noninterference can be provably and straightforwardly enforced via a security-type system for a language that includes cryptographic primitives and message passing. The type system is amenable to extensions, including integrity and public-key cryptography, which makes it attractive for developing secure implementations of non-trivial cryptographic protocols. We plan to explore a semantic justification of these extensions, crystallizing guarantees provided by the typing rules, and to consider cases studies in which it is critical to achieve these guarantees.

Acknowledgments We wish to thank Martín Abadi and Peeter Laud for helpful comments. This work was supported, in part, by the Swedish Research Council and, in part, by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, September 1999.
2. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. of Cryptology*, 15(2):103–127, 2002.
3. A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. Technical report, Chalmers University of Technology, June 2006. Located at `http://www.cs.chalmers.se/~aaskarov/sas06full.pdf`.
4. A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, pages 197–221. Springer-Verlag, September 2005.
5. M. Bellare, A. Desa, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology- Crypto 98*, volume 1462 of *LNCS*, pages 26–46, January 1998.
6. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt 2000*, volume 1976 of *LNCS*, pages 531–545, January 2000.
7. J. Black, P. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 62–75. Springer-Verlag, August 2002.
8. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.
9. S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.
10. T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *Proc. IEEE Computer Security Foundations Workshop*, pages 170–186, 2003.
11. 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.
12. M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. IEEE Computer Security Foundations Workshop*, pages 233–244, July 2000.
13. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

14. D. Duggan. Cryptographic types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 238–252, June 2002.
15. P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.
16. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
17. S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.
18. A. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. CONCUR'05*, number 3653 in LNCS, pages 186–201. Springer-Verlag, August 2005.
19. B. Hicks, D. King, and P. McDaniel. Declassification with cryptographic functions in a security-typed language. Technical Report NAS-TR-0004-2005, Network and Security Center, Department of Computer Science, Pennsylvania State University, May 2005.
20. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 77–91. Springer-Verlag, April 2001.
21. P. Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 159–173. Springer-Verlag, April 2003.
22. P. Laud and V. Vene. A type system for computationally secure information flow. In *Proc. Fundamentals of Computation Theory*, volume 3623 of *LNCS*, pages 365–377, August 2005.
23. P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 112–121, November 1998.
24. D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symp. on Security and Privacy*, pages 177–186, May 1988.
25. J. C. Mitchell. Probabilistic polynomial-time process calculus and security protocol analysis. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 23–29. Springer-Verlag, April 2001.
26. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 2006. To appear.
27. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
28. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.
29. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.
30. C. E. Shannon. A mathematical theory of communication. *Bell System Tech. J.*, 27:623–656, 1948.
31. E. Sumii and B. Pierce. Logical relations for encryption. In *Proc. IEEE Computer Security Foundations Workshop*, pages 256–269, June 2001.
32. D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.
33. D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, January 2000.
34. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

On the Computational Soundness of Cryptographically Masked Flows

Peeter Laud

Tartu University, Institute of Computer Science, and Cybernetica AS

peeter.laud@ut.ee

Abstract

To speak about the security of information flow in programs employing cryptographic operations, definitions based on computational indistinguishability of distributions over program states have to be used. These definitions, as well as the accompanying analysis tools, are complex and error-prone to argue about. *Cryptographically masked flows*, proposed by Askarov, Hedin and Sabelfeld, are an abstract execution model and security definition that attempt to abstract away the details of computational security. This abstract model is useful because analysis of programs can be conducted using the usual techniques for enforcing non-interference.

In this paper we investigate under which conditions this abstract model is computationally sound, i.e. when does the security of a program in their model imply the computational security of this program. This paper spells out a reasonable set of conditions and then proposes a simpler abstract model that is nevertheless no more restrictive than the cryptographically masked flows together with these conditions for soundness.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Operational Semantics, Program Analysis

General Terms Languages, Security, Verification

Keywords Secure Information Flow, Encryption, Computational Soundness, Cryptographically Masked Flows

1. Introduction

Non-interference (Goguen and Meseguer 1982) is the usual way of defining the secure information flow in programs (Sabelfeld and Myers 2003). It states that varying only the secret inputs of the program must not change the public outputs — public outputs are determined by the non-secret inputs only. For programs containing encryption or other cryptographic operations, such a definition may be too strong, because a ciphertext still depends on the plaintext used to produce it, albeit in a manner that cannot be exploited by an adversary that uses only a reasonable amount of resources. Instead, the notion of computational non-interference (Laud 2003) has to be used. The definition of computational non-interference is quite complex in its structure and in the used domains. The usage of such a definition requires the semantics of the program to be probabilistic, making it more difficult to argue about. Also, the precise program analyses based directly on computational non-interference (Laud 2001, 2003; Laud and Vene 2005) tend to have a complex structure and their correctness is not always so obvious.

It would be nice to have a more "abstract" definition for the semantics of the programming language, as well as for the security of the information flow, such that

(i) the used domains and the structure of definitions are more conventional;

(ii) the security of the program in the abstract model would imply its security in the computational model;

(iii) the abstraction would hide the cryptographic details of the encryption (including the necessary use of probabilistic domains), but not much else of the computational model.

Cryptographically masked flows by Askarov et al. (2006) aims to be such a more abstract model. This model considers an imperative programming language (in the original paper this language is quite feature-rich; we consider a stripped-down version of it) with key generation, encryption and decryption as distinguished operations. In the concrete semantics, corresponding to the real-world implementations of the language, the encryption operation is probabilistic (otherwise it cannot be sufficiently secure). The semantics of a program maps the initial state S^{i} to a probability distribution D^{f} over final states. In the model of cryptographically masked flows, henceforth called the *abstract semantics*, the encryption operation is non-deterministic — the encryption algorithm works in the same way as in the concrete semantics, but each time it flips a coin it gets both 0 and 1 as the result. The abstract semantics of a program maps the initial state S^{i} to a set $\mathbf{S}^{\mathrm{f}}$ of possible final states. The set $\mathbf{S}^{\mathrm{f}}$ can be obtained from the distribution D^{f} by just forgetting the probabilities (at least if there are no other probabilistic operations except key generation and encryption; such requirement is put forth by Askarov et al. (2006)). The definition of secure information flow in the setting of cryptographically masked flows is the conventional possibilistic non-interference (Smith and Volpano 1998) stating that the set of the low-slices of possible final states may not depend on the initial secrets. However, when considering whether the low-slices of two states are equal, we sometimes allow the values of the variables containing ciphertexts to differ. The equivalence of low-slices is defined so that generally all ciphertexts are considered equal, but we can distinguish a pair of two different ciphertexts from a pair of equal ciphertexts. Cryptographically masked flows can hence be said to satisfy the objectives (i) and (iii). The aim of the current paper is to investigate, to what extent and under which conditions the objective (ii) is satisfied.

Askarov et al. (2006) also give a type system for checking whether a program satisfies the non-interference property given by the cryptographically masked flows. In the current paper, we do

POPL'08, January 7–12, 2008, San Francisco, California, USA.

not treat this type system in any way; we are interested strictly only in the cryptographically masked flows. Still, the results of this paper and Askarov et al. (2006) together establish that if a program is typable according to this type system, and also satisfies certain conditions that we put on it in this paper, then this program has computationally secure information flow.

In the following we will give a precise definition of the programming language and its concrete and abstract semantices in Sec. 2. We continue by formally stating the definitions of secure information flow in both the concrete and abstract settings in Sec. 3. In Sec. 4 we state and discuss the security definitions that the encryption systems employed in this paper must satisfy. In Sec. 5 we give several examples of programs that seem to violate the computational soundness of cryptographically masked flows and outline the conditions that would exclude such programs, these conditions are formally stated in Sec. 6 and their sufficiency is proved in Sec. 7. Reflecting on the constraints put on the programs we devise a new model for abstract execution and non-interference that is simpler and less restrictive; we describe it in Sec. 8. We finish with a review of some related work in Sec. 9 and discussion on desired properties of such abstractions in general in Sec. 10.

2. Programming language

In this paper we consider programs P in the usual WHILE-language defined by

$$P ::= x := o(x_1, \ldots, x_k) \mid \mathit{skip} \mid P_1 \,;\, P_2 \\ \mid \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \mid \mathit{while}\ b\ \mathit{do}\ P'$$

where $b, x, x_1, \ldots, x_k$ are variables from a given set $\mathbf{Var}$ and o ranges over a fixed set of arithmetic, relational, boolean, etc. operations. Among the operations of the language we will handle the following ones in a special way: new key generation newkey, (symmetric) encryption enc, decryption dec, pairing $(,)$ and projections π_1 and π_2. The language in (Askarov et al. 2006) is much richer, but the subset we are considering here represents the underlying problem well. As usually done in this area (research on secure information flow), we consider only terminating programs — the issue of leaking information by non-termination (or by execution time) is orthogonal to the issues considered here and can be mitigated by known methods (Agat 2000).

Askarov et al. (2006) give a big-step operational semantics for the programming language. The semantics is quite typeful — values from different sources have different types (key, ciphertext, pair, integer) and if the arguments of the operations are not of the right type, the operation gets stuck (in some sense, by disallowing type errors, the semantics already contains some aspects of the enforcement of non-interference). In this paper, we have tried to simplify the operational semantics as much as possible, leaving out such constraints. Rushing ahead, those constraints actually turn out to be necessary for the soundness result, thus they'll appear again in Sec. 6.

The abstract semantics of expressions is given in Fig. 1. The semantics of an n-ary "normal" operation o is a polynomial-time computable function $[\![o]\!] : \mathbf{Val}^n \to \mathbf{Val}$ where $\mathbf{Val} = \{0,1\}^*$ is the set of values (no operations are probabilistic, except key generation and encryption). We assume that there is a distinguished value $\bot \in \mathbf{Val}$ denoting failure, and that all operations are strict with respect to $\bot$. For giving semantics to pairing and projections, we assume that an easily computable and reversible injective function $\rho : \mathbf{Val}^2 \to \mathbf{Val}$ is fixed; this function is the semantics of the pairing operation. For giving semantics to the key generation, encryption and decryption operations, we fix an encryption system $(\mathcal{K}, \mathcal{E}, \mathcal{D})$. Here $\mathcal{K}$ is the key generation algorithm, $\mathcal{E}$ is the encryption and $\mathcal{D}$ the decryption algorithm. The algorithms $\mathcal{K}$ and $\mathcal{E}$ are

$$\frac{M(x) = v}{\langle M, x\rangle \Downarrow^{\mathrm{a}} v}$$

$$\frac{\langle M, e_i\rangle \Downarrow^{\mathrm{a}} v_i \quad [\![o]\!](v_1, \ldots, v_k) = v}{\langle M, o(e_1, \ldots, e_k)\rangle \Downarrow^{\mathrm{a}} v}$$

$$\frac{\langle M, e\rangle \Downarrow^{\mathrm{a}} u \quad \rho(v_1, v_2) = u}{\langle M, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} v_i}$$

$$\frac{\langle M, e\rangle \Downarrow^{\mathrm{a}} u \quad u \notin \mathrm{Dom}\ \rho}{\langle M, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} \bot}$$

$$\frac{v \in \mathrm{Supp}\, \mathcal{K}()}{\langle M, \mathsf{newkey}\rangle \Downarrow^{\mathrm{a}} v}$$

$$\frac{\langle M, e_k\rangle \Downarrow^{\mathrm{a}} v_k \quad \langle M, e_x\rangle \Downarrow^{\mathrm{a}} v_x \quad v_y \in \mathrm{Supp}\, \mathcal{E}(v_k, v_x)}{\langle M, \mathsf{enc}(e_k, e_x)\rangle \Downarrow^{\mathrm{a}} v_y}$$

Figure 1. Abstract semantics of expressions

probabilistic, $\mathcal{D}$ is deterministic. The algorithm $\mathcal{K}$ takes no arguments, the algorithms $\mathcal{E}$ and $\mathcal{D}$ take two — the key and the plain-/ciphertext. For all keys k that can be output by $\mathcal{K}$, for all plaintexts $x \in \mathbf{Val}$ and all ciphertexts y that can be output by $\mathcal{E}(k, x)$, the equality $\mathcal{D}(k, y) = x$ must hold. The decryption is allowed to fail, it has to produce $\bot$ then. The semantics of the operation dec is $\mathcal{D}$. For a probability distribution D we denote by $\mathrm{Supp}\ D$ the set of all such x where $D(x) > 0$.

The ability of operations, particularly decryption, to fail is different from (Askarov et al. 2006). In their treatment, the failures are invisible — the executions where an operation is about to fail just get stuck and do not contribute anything to the final set of states. In our opinion this is unrealistic, as the failures are definitely visible in the real world.

In Fig. 1, M is a *memory* — a mapping from variables to values. We see that $\Downarrow^{\mathrm{a}}$ is non-deterministic — the key generations and encryptions may have several possible values. Compared to the treatment of Askarov et al. (2006), we have made a simplification — they let the program state also to contain the stream of yet-to-be-generated keys and the operation newkey takes and returns the first element of that stream (this detail turns out to be important in the security definition, we will discuss it in Sec. 5).

Having defined the abstract semantics of expressions $\Downarrow^{\mathrm{a}}$ we now define the transition relation of the abstract small-step structural operational semantics $\xrightarrow{\mathrm{a}}$ of the programming language. The relation $\xrightarrow{\mathrm{a}}$ relates program configurations $\langle M, P\rangle$ to program configurations or memories. The definition of $\xrightarrow{\mathrm{a}}$ is completely standard (Nielson and Nielson 1992, Chap. 2.2) and is omitted. But note that the non-determinism of $\Downarrow^{\mathrm{a}}$ also causes the non-determinism of $\xrightarrow{\mathrm{a}}$. We also define $\langle M, P\rangle \overset{\mathrm{a}}{\Longrightarrow} \mathcal{M}$ if $\mathcal{M} = \{M' \mid \langle M, P\rangle \xrightarrow{\mathrm{a}}{}^{*} M'\}$.

On the concrete / computational side, the non-deterministic constructs are replaced by probabilistic ones. We define

- $\Downarrow^{\mathrm{c}}$, relating an expression e to the probability distribution of its values;
- the transition relation $\xrightarrow{\mathrm{c}}$ of the concrete small-step structural operational semantics, relating a program configuration $\langle M, P\rangle$ to a probability distribution D over memories or to a pair $\langle D, P'\rangle$.

(these straightforward definitions are omitted) Given $\xrightarrow{\mathrm{c}}$ we can define the probability of a sequence $\langle M_0, P_0\rangle \to \langle M_1, P_1\rangle \to \cdots \to \langle M_{n-1}, P_{n-1}\rangle \to M_n$ as the product of the probabilities

$p_1, \ldots, p_n$ where p_i is the probability of M_i in the distribution D_i and D_i is given by $\langle M_{i-1}, P_{i-1}\rangle \xrightarrow{c} \langle D_i, P_i\rangle$ (for $1 \le i \le n-1$) or $\langle M_{n-1}, P_{n-1}\rangle \xrightarrow{c} D_n$. Finally we define $\langle M, P\rangle \stackrel{c}{\Longrightarrow} D$ relating the initial program memory M and the program P to the probability distribution D over final program memories. Here the probability assigned by D to some memory M' is the sum of probabilities of all sequences $\langle M, P\rangle \to \cdots \to M'$. The mapping D is indeed a probability distribution (i.e. all probabilities assigned by it sum up to 1) because we have disallowed the non-termination of P.

3. Security definition

Let $\mathbf{Var}_\mathrm{S}, \mathbf{Var}_\mathrm{P} \subseteq \mathbf{Var}$ be the sets of initial secret and final public variables — we want the adversary that learns the final values of the variables in $\mathbf{Var}_\mathrm{P}$ to be unable to deduce anything it did not know before about the initial values of the variables in $\mathbf{Var}_\mathrm{S}$. We demand $\mathbf{Var}_\mathrm{S} \cap \mathbf{Var}_\mathrm{P} = \emptyset$, but not $\mathbf{Var}_\mathrm{S} \cup \mathbf{Var}_\mathrm{P} = \mathbf{Var}$ — there may be auxiliary variables that do belong to neither $\mathbf{Var}_\mathrm{S}$ nor $\mathbf{Var}_\mathrm{P}$. We demand that the program does not use the initial values of such auxiliary variables.

To give the definition of non-interference in the abstract setting, Askarov et al. (2006) first defined when two cryptotexts "look the same". For bit-strings y_1, y_2 they defined $y_1 =_\mathrm{E} y_2$ if there exist such r, k_1, k_2, x_1, x_2, such that $y_1 = \mathcal{E}(r; k_1, x_1)$ and $y_2 = \mathcal{E}(r; k_2, x_2)$. Here the argument r denotes the choice of the random coins for the algorithm $\mathcal{E}$. I.e. $y_1 =_\mathrm{E} y_2$ if they could be ciphertexts generated with the same random coins (initial vectors).

This relaxed equality is applied only to ciphertexts. To use it for the values of program variables we have to fix which of those variables contain ciphertexts. Hence let $\mathbf{Var}_\mathsf{E} \subseteq \mathbf{Var}_\mathrm{P}$ be the public variables that are assumed to contain ciphertexts. For program memories M_1 and M_2 we define the *low-equivalence* of M_1 and M_2, denoted $M_1 \sim_\mathrm{P} M_2$, if $M_1(x) = M_2(x)$ for all $x \in \mathbf{Var}_\mathrm{P} \backslash \mathbf{Var}_\mathsf{E}$ and $M_1(x) =_\mathrm{E} M_2(x)$ for all $x \in \mathbf{Var}_\mathsf{E}$.

Askarov et al. (2006) define a program P to be non-interfering if for all program memories M_1, M_2, such that $M_1 \sim_\mathrm{P} M_2$ and for all memories M_1', such that $\langle M_1, P\rangle \xrightarrow{a}{}^* M_1'$ there exists a memory M_2', such that $\langle M_2, P\rangle \xrightarrow{a}{}^* M_2'$ and $M_1' \sim_\mathrm{P} M_2'$. This definition gives us the standard nondeterministic non-interference, only the definition of equality for values has been changed.

In the computational setting, the non-interference is defined as the computational independence of secret inputs and public outputs. To define the asymptotic computational notions we need a security parameter n relative to which the asymptotics are taken. Hence let the semantics of the operations be parametrized by this security parameter and let their running time be polynomial with respect to this parameter. In particular, the algorithms $\mathcal{K}$, $\mathcal{E}$ and $\mathcal{D}$ take the security parameter as an extra argument. The semantic relations $\Downarrow^c$, $\xrightarrow{c}$ and $\stackrel{c}{\Longrightarrow}$ are thus parametrized with n, too.

Let D be the probability distribution of initial memories for the program P; the adversary knows this distribution. Actually, because of the security parameter, $D = \{D_n\}_{n\in\mathbb{N}}$ is a family of probability distributions over program memories. The program P is computationally non-interferent with respect to D (Laud 2003) if the families of probability distributions (parametrized by n)

$$\{\!|(M|_{\mathbf{Var}_\mathrm{S}}, T|_{\mathbf{Var}_\mathrm{P}}) \mid M \leftarrow D_n, \langle M, P\rangle \stackrel{c}{\Longrightarrow}_n \hat{D}, T \leftarrow \hat{D}|\!\}$$

and

$$\{\!|(M'|_{\mathbf{Var}_\mathrm{S}}, T|_{\mathbf{Var}_\mathrm{P}}) \mid M, M' \leftarrow D_n, \langle M, P\rangle \stackrel{c}{\Longrightarrow}_n \hat{D}, T \leftarrow \hat{D}|\!\}$$

are computationally indistinguishable. Here $\{\!|E \mid C|\!\}$ denotes the distribution of the random expression E under the conditions C. The notation $M \leftarrow D_n$ means that the random variable is distributed according to D_n. Hence the first of the above distributions is that of the initial values of the secret variables and final values of the public variables, where the initial memory M is sampled according to D_n and the final memory T is obtained by executing the program P (which is probabilistic) on M. The second of the above distributions is that of the same values of the same variables, but here the initial memory M' and the final memory T correspond to different runs (the initial memories M and M' are sampled independently of each other). Two families of probability distributions $D = \{D_n\}_{n\in\mathbb{N}}$ and $D' = \{D'_n\}_{n\in\mathbb{N}}$ are *computationally indistinguishable* (this is the cryptographic equivalent for being "the same") if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ the difference

$$\Pr[\mathcal{A}(n, x) = 1 \mid x \leftarrow D_n] - \Pr[\mathcal{A}(n, x) = 1 \mid x \leftarrow D'_n]$$

is negligible in n. Here $\mathcal{A}(n, x)$ denotes the probability distribution of the outputs of the algorithm $\mathcal{A}$ on the input (n, x). A function f is negligible if f is $o(1/p)$ for any polynomial p.

Our desired result We want to have a soundness theorem with more or less the following wording:

> Let P be a program. If P satisfies certain conditions and the initial probability distribution D satisfies certain conditions and P is non-interferent in the abstract setting then P is computationally non-interferent with respect to the initial distribution D.

Here the conditions on D should be something natural, for example the independence of the secret values from the public ones. The conditions on P should be verifiable in the abstract setting, otherwise we lose the modularity of the approach. In the Sec. 5 we take a look at what these conditions could be. Let us call a program P *well-structured* if it obeys those conditions.

In the following we often have to speak about the public part of the result of some computation. For a memory M we thus define M_P as the restriction of M to $\mathbf{Var}_\mathrm{P}$. For a set of memories $\mathcal{M}$ we define $\mathcal{M}_\mathrm{P} = \{M_\mathrm{P} \mid M \in \mathcal{M}\}$. For a probability distribution D over memories we have to collapse all memories with the same public part — we define $D_\mathrm{P}(\bar{M}) = \sum_{M:M_\mathrm{P}=\bar{M}} D(M)$.

4. Security of encryption systems

So far we have only defined the functionality of an encryption system, but not its security. This definition is necessary when arguing about the security of information flow. Obviously, we require that our encryption system hides the contents of plaintexts. To simplify our arguments in the rest of the paper we also want the *length* of the plaintext to be hidden. We still need more — the encryption must also hide the *identities* of plaintexts. This means that it must be impossible to determine whether two ciphertexts were generated using the same key or different keys. The programs in our programming language are able to decrypt, too, hence we also need to protect the integrity of encrypted messages. We do not want to put many constraints on the programs, hence the security properties must hold even if the keys are treated quite arbitrarily (short of leaking them). In particular, it must be possible to treat the keys as parts of plaintexts, too. The details are given below. There we also discuss the existence of such encryption systems.

Against passive attacks, the most common security definition is the indistinguishability under chosen plaintext attacks (IND-CPA security) (Bellare et al. 1997). It states that there exists no PPT adversary $\mathcal{A}$ (that has access to an oracle), such that the difference of probabilities

$$\Pr[\mathcal{A}^{\mathcal{E}_n(k,\cdot)}(n) = 1 \mid k \leftarrow \mathcal{K}_n()] - \Pr[\mathcal{A}^{\mathcal{E}_n(k,\mathbf{0}^{|\cdot|})}(n) = 1 \mid k \leftarrow \mathcal{K}_n()]$$

is non-negligible. This definition is a typical instance of security definitions of cryptographic primitives and systems where the indistinguishability of the "real" functionality from the "ideal" functionality is required. Here the real functionality is the encryption functionality — given a plaintext it returns a corresponding ciphertext. The ideal functionality also returns a ciphertext, but this ciphertext is generated without actually using the plaintext (except its length).

In the definition of $=_{\mathrm{E}}$, ciphertexts created with different keys may also be considered equal. Hence we need the encryption system to hide the identities of keys (Abadi and Rogaway 2000) as well: for no PPT adversary $\mathcal{A}$, the following difference may be non-negligible:

$$\Pr[\mathcal{A}^{\mathcal{E}_n(k,\cdot),\mathcal{E}_n(k',\cdot)}(n) = 1 \mid k, k' \leftarrow \mathcal{K}_n()] - \Pr[\mathcal{A}^{\mathcal{E}_n(k,\cdot),\mathcal{E}_n(k,\cdot)}(n) = 1 \mid k \leftarrow \mathcal{K}_n()] \ .$$

Our security proofs in this paper will be simpler if we do not state that in the ideal functionality all keys are the same, but state that in the ideal functionality all keys are different. We also give the adversary the possibility to choose among several encryption oracles, not just two (in the definition of IND-CPA, the adversary could also be allowed to access several encrypting oracles simultaneously).

Let $\mathcal{O}$ be an oracle that works as follows:

- At the initialization (or before it answers its very first query) it independently generates the keys k_i using the algorithm $\mathcal{K}_n$ for all $i \in \mathbb{N}$. Actually, the keys k_i are not all generated in the beginning of the run, but only right before they are first used.
- When queried with (i, x) the machine $\mathcal{O}$ returns $\mathcal{E}_n(k_i, x)$.

Let $\mathcal{O}'$ be an oracle that on query (i, x) generates a new key k, encrypts x with it and returns the result. An encryption system hides the identities of the keys if and only if no PPT adversary is able to distinguish the oracles $\mathcal{O}$ and $\mathcal{O}'$ with non-negligible advantage.

The non-interference definition in the abstract model does not attempt to rule out *key cycles* (Abadi and Rogaway 2000) in any way. A key cycle of length 1 occurs when a key k (or a message where k may be obtained from) is used as a plaintext in encryption with k. In a longer key cycle, the key k_1 is encrypted with k_2, the key k_2 is encrypted with k_3, etc., until the key k_n is encrypted again with k_1. In such scenarios we can find no real functionality, as given in the definition of IND-CPA, that can be replaced with the ideal one. Therefore the definition of IND-CPA does not say anything about the security of such usage of keys. Such usage of keys is certainly possible in programs.

Also, $=_{\mathrm{E}}$ will have no problem relating ciphertexts whose plaintexts are of obviously different lengths. It may make sense to refine $=_{\mathrm{E}}$ so that the lengths of plaintexts were discriminated, but the issue of concealing the lengths is orthogonal to other issues in secure information flow. Hence the current choice should be considered reasonable. A strengthening of the definition of IND-CPA to *key-dependent messages* (Black et al. 2002) that also conceals the lengths of the plaintexts and the identities of the keys can be given as follows.

Let $\mathcal{O}$ be an oracle that works as follows

- At the initialization it independently generates the keys k_i using the algorithm $\mathcal{K}_n$ for all $i \in \mathbb{N}$. Actually, the keys k_i are not all generated in the beginning of the run, but only right before they are first used...
- $\mathcal{O}$ accepts queries of the form (i, e) where $i \in \mathbb{N}$ and e is an expression in some Turing-complete language whose running time is polynomial in n (we may also state that the query contains the maximum running time of e as well). The expression e may contain free variables k_j.
- When queried with (i, e), the machine $\mathcal{O}$ evaluates e, substituting the values of the keys k_j to the free variables k_j of e. It then encrypts the result with the key k_i and returns it.

Let $\mathcal{O}'$ be an oracle that on each query generates a new key and returns the encryption of a fixed constant with this key. The encryption system $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ is *IND-CPA-secure, which-key concealing and length-concealing in presence of key-dependent messages* if no PPT adversary $\mathcal{A}$ can distinguish $\mathcal{O}$ from $\mathcal{O}'$, i.e. the difference of probabilities

$$\Pr[\mathcal{A}^{\mathcal{O}(\cdot)}(n)] - \Pr[\mathcal{A}^{\mathcal{O}'(\cdot)}(n)]$$

must be negligible for all PPT $\mathcal{A}$.

The properties considered above only deal with confidentiality of messages, and only with encryption. They do not state anything about what happens if the decryption algorithm is invoked (the definition of encryption systems states that the decryption of a correctly constructed ciphertext must give back the corresponding plaintext, but does not state anything about decrypting other bit-strings). The property that we are going to need is *plaintext integrity* (INT-PTXT) (Bellare and Namprempre 2000) stating that no PPT adversary $\mathcal{A}$ that is given access to the encryption oracle $\mathcal{E}(k, \cdot)$ (where k is generated using the key generation algorithm) is able (with non-negligible probability) to output a ciphertext c, such that $\mathcal{D}(k, c) = p \neq \perp$ and $\mathcal{A}$ had not queried the encryption oracle with p before. Askarov et al. (2006) argue that if the encryption system has plaintext integrity then $\mathcal{D}(k, \mathcal{E}(k', x))$ results in error almost always (i.e. the opposite has only negligible probability). Here x is any plaintext and k, k' are two keys that are independent of each other.

The above definition of INT-PTXT does not allow encryption cycles, either. We will turn the given definition into one that allows key-dependent messages in the same way as was done for the definition of IND-CPA. First, we give the adversary access to multiple encryption oracles $\mathcal{E}(k_1, \cdot), \mathcal{E}(k_2, \cdot), \ldots$ where the keys k_i are independently generated. Such access is equivalent to the access to an oracle $\mathcal{O}$ that on query (i, x) returns $\mathcal{E}(k_i, x)$. After interacting with the oracle $\mathcal{O}$, the adversary outputs a pair (i, c) and wins if $\mathcal{D}(k_i, c) = p \neq \perp$ and the adversary did not query $\mathcal{O}$ with (i, p) before. The modification of INT-PTXT we have done so far (replacing a single key k with multiple keys $k_1, k_2, \ldots$) has not been substantial — the modified definition can be proved equivalent to the original definition by a standard hybrid argument (Goldreich 2001, Chap. 3.2.3). We will now modify the definition by allowing key-dependent messages as well — in the query (i, x) to $\mathcal{O}$ the component x is not just a bit-string but an expression for computing the bit-string to be encrypted. As before, x may contain free variables $k_1, k_2, \ldots$, these are substituted with the values of the keys $k_1, k_2, \ldots$ that have been generated by $\mathcal{O}$.

Existence. It is not known how to construct encryption systems that are secure in the presence of key-dependent messages, as long as the construction is in the "plain model" i.e. assumes only the existence of one-way functions. Black et al. (2002) give a construction for IND-CPA-secure (with key-dependent messages) encryption system in the *random oracle model* (Bellare and Rogaway 1993). This model assumes the existence of a globally fixed random function $H : \{0,1\}^* \to \{0,1\}^\omega$ that all parties (including the adversary, and the expressions it sends to its oracle(s)) can access (in practice, H is replaced by some (function based on some) cryptographic hash function). The distribution of H is such that all bits in its image are distributed fairly and independently of each other. In the encryption system by Black et al. (2002), the key generation algorithm just returns an n-bit random string (where n is the se-

curity parameter). The encryption algorithm $\mathcal{E}_n(k,x)$ generates an n-bit random string r and returns $(r, H_{|x|}(k\|r) \oplus x)$ where $\|$ denotes the concatenation of bit-strings and H_ℓ returns the first ℓ bits of H. Black et al. (2002) show this system to be IND-CPA-secure. A slight modification of their proof is sufficient to show that this encryption system also conceals the identities of the keys. This encryption system obviously does not hide the length of the plaintext, padding has to be used for that.

Bellare and Namprempre (2000) show how to use message authentication codes (MACs) to provide plaintext (and ciphertext) integrity for encryption systems. A MAC consists of three algorithms — key generation algorithm, tagging algorithm (taking the key and the message as inputs) and verification algorithm (taking the key, the message and the alleged tag as inputs). The tagging algorithm may be deterministic in which case the verification algorithm simply has to recompute it. A MAC is *weakly unforgeable (WUF)*, if no PPT adversary with the access to tagging and verification algorithms (as oracles) can produce a message and a valid tag, such that the message had not been submitted to the tagging oracle before that. Bellare and Namprempre (2000) prove that a compound encryption system where a message is first tagged with a weakly unforgeable MAC and then encrypted with an IND-CPA-secure cryptosystem (using different keys) provides plaintext integrity. As the encryption operation is the last step in the construction, the construction also preserves the concealing of key identities.

We can modify the definition of weak unforgeability to also allow key-dependent messages (the adversary can then submit expressions, not just messages to the tagging and verification oracles). The proof by Bellare and Namprempre (2000) can then be modified, showing that if the construction is made using primitives that are secure in the presence of key-dependent messages then the resulting encryption system is INT-PTXT-secure also even with key-dependent messages.

A WUF-secure (with key-dependent messages) MAC is easy to construct in the random oracle model. Let the key generation algorithm return a random n-bit string and let the tag of the message x with the key k be the first n bits of $H(k\|x)$. In this way the tags of the messages are really just random bit-strings that do not reveal anything about k.

5. Non-well-structured programs

In this section we give some examples of programs that are secure in the abstract setting, but possibly insecure in the computational setting. We then explain what patterns in programs have to be excluded to make sure that the soundness theorem would not apply to this or analogous programs.

5.1 Bad keys

The program

$$k := s;\ p := \mathsf{enc}(k, s)$$

where $\mathbf{Var}_\mathrm{S} = \{s\}$ and $\mathbf{Var}_\mathrm{P} = \mathbf{Var}_\mathrm{E} = \{p\}$, is secure in the abstract setting. In the computational setting we do not know whether k is a good key. Hence we do not know whether $\mathcal{E}(s,s)$ sufficiently hides s or not. We cannot use the security definitions of encryption systems to argue about the computational security of this program.

The conditions we put on programs must make sure that only keys generated with the operation newkey can be used as keys in encryption operations.

5.2 Constructing ciphertexts

Similarly to using only good keys when encrypting we must use only good ciphertexts when applying the relaxed equivalence $=_\mathrm{E}$ on them.

5.3 Publishing keys

Consider the following program

$$k := \mathsf{newkey};\ p := \mathsf{enc}(k, s)$$

where $\mathbf{Var}_\mathrm{S} = \{s\}$, $\mathbf{Var}_\mathrm{P} = \{k, p\}$ and $\mathbf{Var}_\mathrm{E} = \{p\}$. According to the possibilistic non-interference definition, this program is secure. Indeed, an initial memory $\{s \mapsto v_s\}$ is transformed to the final memories where the value of k is just a key and the part of the value of p that matters for the relation $=_\mathrm{E}$ is just a random initial vector. Hence the public part of the set of final memories does not depend on the initial secret.

On the other hand we would certainly want to consider the preceding program secure if the set of public variables $\mathbf{Var}_\mathrm{P}$ would have consisted of just the variable p. To avoid considering such programs secure we must make sure that the keys cannot become public. This includes using these keys in computations (other than encryption and possibly decryption), such that the results of those computations become public.

Askarov et al. (2006) actually consider the previous program to be insecure because the program state also contains the values of all yet-to-be-generated keys; these values are treated as high-security inputs. Hence one cannot publish keys or anything that depends on them (except when used in encryption).

5.4 Possibilistic vs. probabilistic security

The following well-known example (McLean 1990) shows that the possibilistic non-interference does not always imply probabilistic (and similarly computational) non-interference.

$$x := \mathsf{rnd}(0,1);\ \mathit{if}\ x\ \mathit{then}\ l := h\ \mathit{else}\ l := \mathsf{rnd}(1,100)$$

where $\mathsf{rnd}(a,b)$ returns a uniformly chosen random integer between a and b, $l \in \mathbf{Var}_\mathrm{P}$, $h \in \mathbf{Var}_\mathrm{S}$ and it is known that the value of h is between 1 and 100. In the possibilistic setting the semantics of $\mathsf{rnd}(a,b)$ is nondeterministic, we have $\mathsf{rnd}(a,b) \Downarrow^\mathrm{a} c$ for all c where $a \le c \le b$.

In the possibilistic setting, the above program is secure because the set of possible final values of l does not depend on the value of h. In the probabilistic setting it is insecure because the most probable final value of l is equal to the value of h. In our programming language, we can implement rnd (or something close to it) by using the probabilistic operation enc. For certain constructions of secure encryption systems, the output of $\mathcal{E}$ is indistinguishable from a uniformly distributed bit-string.

To avoid considering the previous program, we must disallow computations with ciphertexts. The ciphertexts may be published, included in pairs, encrypted or decrypted, but their "actual value" may not be used in computations.

5.5 Non-failure of bad decryptions

Consider the following program:

$$k := \mathsf{newkey};\ k' := \mathsf{newkey};\ x := \mathsf{enc}(k, C);\ y := \mathsf{dec}(k', x);\\ \mathit{if}\ y = \bot\ \mathit{then}\ l := h\ \mathit{else}\ l := 1 - h$$

where C is some constant, $\mathbf{Var}_\mathrm{P} = \{l\}$, $\mathbf{Var}_\mathrm{S} = \{h\}$ and the possible values of h are 0 and 1. In the concrete (probabilistic) setting, the program is insecure because the *then*-branch is taken with overwhelming probability.

The model of cryptographically masked flows (Askarov et al. 2006) makes the assumption that the decryption with the wrong key always fails. I.e. if $k \neq k'$ then $\mathsf{dec}(k', \mathsf{enc}(k, x))$ returns $\bot$ with probability equal to 1, and not to $1 - \alpha$ for some negligible α. Under that assumption the above program is insecure, too, because the *else*-branch is never taken.

However, it is not clear whether there exist (even in the random oracle model) encryption systems with such a property, that also

satisfy other properties we stated in Sec. 4, in particular hiding the identity of keys. If we drop the requirement that the decryption with the wrong key always fails (i.e. $\mathsf{dec}(k', \mathsf{enc}(k, x))$ returns $\bot$ with probability of only $1 - \alpha$) then, in the possibilistic setting, it is actually possible to take the *else*-branch in the above program, which is then defined as secure.

Note that this particular program is actually insecure by the definitions given by Askarov et al. (2006) even if we drop the requirement that the decryption with a wrong key always fails, because at the failure of the decryption the execution becomes stuck and hence the *then*-branch is unreachable. But then we could just replace the predicate "$y = \bot$" with some other predicate P that is satisfied by some of the possible values of y and not satisfied by others. Both branches would then still be reachable in the abstract semantics, but in the concrete semantics the branch corresponding to the value of $P(\bot)$ would be chosen most of the time.

There is also a negligible chance that the two key generations in the considered program return the same key. If this happens then the value of h can also be deduced from the value of l because the *else*-branch is always taken.

We do not have a good remedy against cases like this where the abstract semantics considers the events that happen only negligibly often in the concrete semantics as equals to the events that happen almost always in the concrete semantics. We thus resort to modifying the abstract semantics, such that two key generations cannot produce equal keys, and the decryption of a ciphertext is possible only with the key that was used in creating it. We change the definition of $\Downarrow^{\mathrm{a}}$, it is now a relation of the form $\langle M, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle v, K_\bullet, C_\bullet\rangle$ where M is the memory, e the evaluated expression and v its value. Additionally, $K_\circ$ and $K_\bullet$ are the sets of generated keys before and after the evaluation of e, and $C_\circ/C_\bullet$ is the set of triples (c, k, p) of ciphertexts, keys and plaintexts, such that c has been generated by encrypting p with k during the course of the program. The definition of $\Downarrow^{\mathrm{a}}$ for particular expressions is changed as follows:

- For key generations, we have the side condition that the newly generated key v is not a member of $K_\circ$. The set $K_\bullet$ is then defined as $K_\circ \cup \{v\}$ (for all other operations, $K_\bullet = K_\circ$).
- For encryptions, the generated ciphertext may not yet be a ciphertext in the set $C_\circ$. The triple of the newly generated ciphertext, key and plaintext is added to $C_\bullet$ along with the rest of $C_\circ$ (for all other operations, $C_\bullet = C_\circ$).
- The decryption operation $\mathsf{dec}(k, c)$ first computes the plaintext $v_p = \mathcal{D}(M(k), M(c))$. It then checks whether $(v_c, M(k), v_p)$ belongs to $C_\circ$ for some value v_c. If this is the case then $\mathsf{dec}(k, c)$ returns v_p, otherwise it returns $\bot$.

The sets K and C are also added to program configurations.

6. Well-structured programs

To satisfy the necessary constraints demonstrated by the programs in the previous section, it is sufficient to require that all operations that the program (in the abstract semantics) performs are well-typed. Here the types $\tau \in \mathbf{T}$ are defined by the following grammar:

$$\tau ::= \mathit{int} \mid \mathit{key} \mid \mathit{enc}(\tau) \mid (\tau, \tau),$$

and the operations of the program expect and produce the values of the following types:

- arithmetic operations (of arity k): $\mathit{int}^k \to \mathit{int}$;
- key generation: key;
- encryption: $\mathit{key} \times \tau \to \mathit{enc}(\tau)$;
- decryption: $\mathit{key} \times \mathit{enc}(\tau) \to \tau$;

$$\frac{M(x) = v \quad \Gamma(x) = \tau}{\langle M, \Gamma, K, C, x\rangle \Downarrow^{\mathrm{a}} \langle v, \tau, K, C\rangle}$$

$$\frac{\langle M, \Gamma, K_{i-1}, C_{i-1}, e_i\rangle \Downarrow^{\mathrm{a}} \langle v_i, \mathit{int}, K_i, C_i\rangle \quad [\![o]\!](v_1, \ldots, v_k) = v}{\langle M, \Gamma, K_0, C_0, o(e_1, \ldots, e_k)\rangle \Downarrow^{\mathrm{a}} \langle v, \mathit{int}, K_k, C_k\rangle}$$

$$\frac{\langle M, \Gamma, K_{i-1}, C_{i-1}, e_i\rangle \Downarrow^{\mathrm{a}} \langle v_i, \tau_i, K_i, C_i\rangle \quad \rho(v_1, v_2) = v}{\langle M, \Gamma, K_0, C_0, (e_1, e_2)\rangle \Downarrow^{\mathrm{a}} \langle v, (\tau_1, \tau_2), K_2, C_2\rangle}$$

$$\frac{\langle M, \Gamma, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle u, (\tau_1, \tau_2), K_\bullet, C_\bullet\rangle \quad \rho(v_1, v_2) = u}{\langle M, \Gamma, K_\circ, C_\circ, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} \langle v_i, \tau_i, K_\bullet, C_\bullet\rangle}$$

$$\frac{\langle M, \Gamma, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle u, (\tau_1, \tau_2), K_\bullet, C_\bullet\rangle \quad u \notin \mathrm{Dom}\ \rho}{\langle M, \Gamma, K_\circ, C_\circ, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} \langle \bot, \tau_i, K_\bullet, C_\bullet\rangle}$$

$$\frac{v \in \big(\mathrm{Supp}\, \mathcal{K}()\big) \backslash K}{\langle M, \Gamma, K, C, \mathsf{newkey}\rangle \Downarrow^{\mathrm{a}} \langle v, \mathit{key}, K \cup \{v\}, C\rangle}$$

$$\frac{\begin{array}{c}\langle M, \Gamma, K_0, C_0, e_k\rangle \Downarrow^{\mathrm{a}} \langle v_k, \mathit{key}, K_1, C_1\rangle \\ \langle M, \Gamma, K_1, C_1, e_x\rangle \Downarrow^{\mathrm{a}} \langle v_x, \tau, K_2, C_2\rangle \\ v_y \in \big(\mathrm{Supp}\, \mathcal{E}(v_k, v_x)\big) \backslash \boldsymbol{\pi}_1(C_2) \quad C_3 = C_2 \cup \{(v_y, v_k, v_x)\}\end{array}}{\langle M, \Gamma, K_0, C_0, \mathsf{enc}(e_k, e_x)\rangle \Downarrow^{\mathrm{a}} \langle v_y, \mathit{enc}(\tau), K_2, C_3\rangle}$$

$$\frac{\begin{array}{c}\langle M, \Gamma, K_0, C_0, e_k\rangle \Downarrow^{\mathrm{a}} \langle v_k, \mathit{key}, K_1, C_1\rangle \\ \langle M, \Gamma, K_1, C_1, e_y\rangle \Downarrow^{\mathrm{a}} \langle v_y, \mathit{enc}(\tau), K_2, C_2\rangle \quad v_p = \mathcal{D}(v_k, v_y) \\ v = \mathbf{if}\ (v_k, v_p) \in \boldsymbol{\pi}_{2,3}(C_2)\ \mathbf{then}\ v_p\ \mathbf{else}\ \bot\end{array}}{\begin{array}{l}\langle M, \Gamma, K_0, C_0, \mathsf{dec}(e_k, e_y)\rangle \Downarrow^{\mathrm{a}} \langle v, \tau, K_2, C_2\rangle \\ \text{where } \boldsymbol{\pi}_{i_1,\ldots,i_k}(C) \text{ denotes } \{(u_{i_1}, \ldots, u_{i_k}) \mid (u_1, u_2, u_3) \in C\}\end{array}}$$

Figure 2. Extended $\Downarrow^{\mathrm{a}}$

- pairing: $\tau_1 \times \tau_2 \to (\tau_1, \tau_2)$;
- projections: $(\tau_1, \tau_2) \to \tau_i$.

Also, the guards of the branching statements must have the type int, the secret inputs (i.e. the initial values of the variables in $\mathbf{Var}_{\mathrm{S}}$) must also have the type int, final values of the variables in $\mathbf{Var}_{\mathsf{E}}$ must have types of the form $\mathit{enc}(\tau)$, and final values of the variables in $\mathbf{Var}_{\mathrm{P}}$ may not have a type that has a component key (define that the only component of the types int, key and $\mathit{enc}(\tau)$ is that type itself, and the components of the type (τ_1, τ_2) are the components of τ_1 and τ_2).

The previous paragraph spoke about the types of values, not variables. Indeed, we can allow the typing to be *dynamic*, extending the memories with the information about the current types of variables. This dynamic typing disallows the usage of non-keys as keys and the usage of "actual values" of keys (excluding encryption and decryption) and ciphertexts. To formalize this typing we again extend the definition of $\Downarrow^{\mathrm{a}}$: it is now a set of judgements of the form $\langle M, \Gamma, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle v, \tau, K_\bullet, C_\bullet\rangle$, where Γ is a mapping from variables to types and τ is the type of v. The modified $\Downarrow^{\mathrm{a}}$, also covering the modifications of Sec. 5.5, is given in Fig. 2.

The typing Γ is also added to program configurations; the type $\Gamma(x)$ is updated whenever x is assigned to. The extended abstract semantics is given in Fig. 3. We call a program **well-structured** if its execution according to the abstract semantics never gets stuck, i.e. no such configuration is reachable where there is still a program P, but no outgoing transitions. Also, a well-structured program must satisfy the constraints on the final types of variables, as stated above (for variables in $\mathbf{Var}_{\mathrm{P}}$, the type may not be key, and for variables in $\mathbf{Var}_{\mathsf{E}}$, the type must be $\mathit{enc}(\tau)$ for some τ).

The type system still allows us to perform cryptographic operations quite freely, as long as they are not combined with operations

$$\frac{\langle M,\Gamma,K_\circ,C_\circ,e\rangle \Downarrow^{\mathrm{a}} \langle v,\tau,K_\bullet,C_\bullet\rangle}{\langle M,\Gamma,K_\circ,C_\circ,x:=e\rangle \xrightarrow{\mathrm{a}} \langle M[x\mapsto v],\Gamma[x\mapsto\tau],K_\bullet,C_\bullet\rangle}$$

$$\frac{}{\langle M,\Gamma,K,C,\mathit{skip}\rangle \xrightarrow{\mathrm{a}} \langle M,\Gamma,K,C\rangle}$$

$$\frac{\langle M,\Gamma,K,C,P_1\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C'\rangle}{\langle M,\Gamma,K,C,P_1;P_2\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C',P_2\rangle}$$

$$\frac{\langle M,\Gamma,K,C,P_1\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C',P_1'\rangle}{\langle M,\Gamma,K,C,P_1;P_2\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C',P_1';P_2\rangle}$$

$$\frac{\langle M,\Gamma,K_\circ,C_\circ,e\rangle \Downarrow^{\mathrm{a}} \langle b,\mathit{int},K_\bullet,C_\bullet\rangle \quad b\in\{0,1\}}{\langle M,\Gamma,K_\circ,C_\circ,\mathit{if}\ e\ \mathit{then}\ P_1\ \mathit{else}\ P_0\rangle \xrightarrow{\mathrm{a}} \langle M,\Gamma,K_\bullet,C_\bullet,P_i\rangle}$$

$$\frac{}{\langle M,\Gamma,K,C,\mathit{while}\ e\ \mathit{do}\ P\rangle \xrightarrow{\mathrm{a}} \langle M,\Gamma,K,C,\mathit{if}\ e\ \mathit{then}\ (P;\mathit{while}\ e\ \mathit{do}\ P)\ \mathit{else}\ \mathit{skip}\rangle}$$

Figure 3. Instrumented semantics $\xrightarrow{\mathrm{a}}$

that make use of the "actual value" of a key or a ciphertext. We are allowed to copy keys and ciphertexts from one variable to another, as well as to encrypt keys and ciphertexts (or pairs containing a key or a ciphertext as a component). Among the restrictions the inability to use the "actual value" of a ciphertext is the most significant one, but likely unavoidable, because of Sec. 5.4 (otherwise our abstract semantics has to be probabilistic as well). Also, Sec. 5.3 prohibits us making public the values of the keys, hence our type system must ensure that the keys do not end up as values of type int.

LEMMA 1. *Let P be a well-structured program and M a memory. Let $\langle M,\Gamma,K,C,P\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C',P'\rangle$. Then P' is well-structured, too.*

PROOF. Well-structuredness of a program implies the well-structuredness of all of its subprograms. Consider the definition of $\xrightarrow{\mathrm{a}}$. According to it, P' is either a subprogram of P, or $P' \equiv P''; \mathit{while}\ b\ \mathit{do}\ P''$, such that $P \equiv \mathit{while}\ b\ \mathit{do}\ P''$. In the last case P'' and $\mathit{while}\ b\ \mathit{do}\ P''$ are well-structured with respect to the same typing, and their sequential composition admits the same typing as well. □

We can now state the main result of this paper:

THEOREM 2. *If a well-structured program P that does not assign to variables in $\mathbf{Var}_{\mathrm{S}}$ and does not use the initial values of the variables with types different from int has possibilistic non-interference then it has probabilistic non-interference for all families of probability distributions D over initial memories where the variables in $\mathbf{Var}_{\mathrm{S}}$ are independent of the rest of the variables.*

We see that the conditions on P, besides the well-structuredness, are not significant. They can be worked around by adding more variables to P.

7. Security proof

Here we will sketch the proof of Theorem 2. We are going to perform the standard steps in reasoning about programs that contain cryptographic operations — we replace the parts of the program that correspond to the real functionality of the cryptographic primitives with the code that corresponds to the ideal functionality (Sec. 7.1). The modification will transform the program P to some program $\hat{P}$, thereby also changing both the abstract and the concrete semantics of the program. The concrete semantics will change only indistinguishably. We will compare the abstract execution of P with the concrete execution of $\hat{P}$ and conclude that they run in lock-step, producing similar structures of final states (Sec. 7.2). In particular, we will show that for each M, the set of abstract final states $\mathcal{M}$, where $\langle M,P\rangle \stackrel{\mathrm{a}}{\Longrightarrow} \mathcal{M}$, determines the distribution D, where $\langle M,\hat{P}\rangle \stackrel{\mathrm{c}}{\Longrightarrow} D$. If the program P has secure information flow in the abstract setting then the program $\hat{P}$ is probabilistically non-interferent[1]. The concrete semantics of P is indistinguishable from the semantics of $\hat{P}$ for an adversary that does not see the keys generated by the programs, hence P is computationally non-interferent.

7.1 Program modifications

We would like to apply the definition of IND-CPA (including the concealing of key identities and plaintext lengths, and security in presence of key-dependent messages), thereby changing the expressions $\mathsf{enc}(k,y)$ to expressions $\mathsf{enc}(\mathsf{newkey},0)$. But we cannot apply this transformation right away — in the definition of IND-CPA the encryption keys may only be used in encryption operations (and they may also be used in arbitrary manner to create the plaintexts that are encrypted), but in our program they may also be used in decryption operations. We have to apply the definition of INT-PTXT first.

The well-structuredness of programs ensures that we only attempt to decrypt valid ciphertexts. We are not ensured that the key used for decryption is the right one. The plaintext integrity of the used encryption system guarantees that if the used key is not the correct one then the decryption almost always fails in the concrete semantics. The modifications made to the abstract semantics in Sec. 5.5 also ensure this for the abstract semantics. We can modify the program so that we record the plaintext of each ciphertext, as well as (the identity of) the key used to create it, and replace the decryption with the return of the plaintext if the comparison of key identities succeeds. This modification does not change the abstract semantics of a program and changes its concrete semantics only indistinguishably.

Let P_0 be the original program. We perform the following program modifications. To simplify the presentation we assume that there are no nested expressions, i.e. in each statement $x := e$ in the program the expression e contains just a single operation. We also assume that all guard expressions in if- and $while$-statements are just variables. First, we introduce a new variable idk of type int for producing key names and prepend the program with the initialization $idk := 0$. Then we rewrite all assignments and branches according to Table 1.

We see that in the modified program all variables contain pairs whose first component is the original value of the variable. The second component records the auxiliary information for eliminating decryptions:

- for keys, we record their identity;
- for ciphertexts, we record the identity of the key and the plaintext with associated auxiliary information;
- for pairs, we record the auxiliary information of both components;
- we do not record anything for integers, but still keep the second component to avoid special cases in Table 1.

To finish the execution of the program with the same values of variables in $\mathbf{Var}_{\mathrm{P}}$ as before we append to the program the statements

[1] The security of $\hat{P}$ is even information-theoretic (Sabelfeld and Sands 1999, Sec. 5), not just computational

before	after
$x := y$	$x := y$
$x := o(x_1, \ldots, x_k)$	$x := (o(\pi_1(x_1), \ldots, \pi_1(x_k)), 0)$
$x := \mathsf{newkey}$	$idk := idk + 1;$ $x := (\mathsf{newkey}, idk)$
$x := \mathsf{enc}(k, y)$	$x := (\mathsf{enc}(\pi_1(k), \pi_1(y)), (\pi_2(k), y))$
$y := \mathsf{dec}(k, x)$	*if* $\pi_1(\pi_2(x)) = \pi_2(k)$ *then* $\pi_2(\pi_2(x))$ *else* $\perp$
$x := (y, z)$	$x := ((\pi_1(y), \pi_1(z)), (\pi_2(y), \pi_2(z)))$
$y := \pi_i(x)$	$y := (\pi_i(\pi_1(x)), \pi_i(\pi_2(x)))$
if $b \ldots$ / *while* $b \ldots$	*if* $\pi_1(b) \ldots$ / *while* $\pi_1(b) \ldots$

Table 1. Removing decryption operations

$x := \pi_1(x)$ for all $x \in \mathbf{Var}_{\mathrm{P}}$. We also must introduce the second component to all initial values of the variables. For all variables x whose initial values are used by the program we prepend $x := (x, 0)$ to the program (By Thm. 2, all those variables have the initial type int). Let P be the resulting program. Let P' be the program P without the final statements $x := \pi_1(x)$ for $x \in \mathbf{Var}_{\mathrm{P}}$.

LEMMA 3. *If P_0 is a well-structured program then so is P. Moreover, for each initial state the final typings Γ and Γ' of P_0 and P' are such, that $\Gamma'(x) = (\Gamma(x), \ldots)$ for all $x \in \mathbf{Var}$.*

We have introduced the relation $=_{\mathrm{E}}$ on values — two typed values are related by $=_{\mathrm{E}}$ if they are equal or if they are both ciphertexts and have the same initial vector. We now define a more relaxed version of it: we say that $v =_{\mathrm{EK}} v'$ if $v =_{\mathrm{E}} v'$ or both v and v' are keys.

LEMMA 4. *Let M_0 be an initial state. If $\langle M_0, \lambda x.int, \emptyset, \emptyset, P_0\rangle \stackrel{\mathrm{a}}{\Longrightarrow} \mathcal{M}$ and $\langle M_0, \lambda x.int, \emptyset, \emptyset, P'\rangle \stackrel{\mathrm{a}}{\Longrightarrow} \mathcal{M}'$ then for all $M \in \mathcal{M}$ there exists some $M' \in \mathcal{M}'$, and for all $M' \in \mathcal{M}'$ there exists some $M \in \mathcal{M}$, such that $M(x) =_{\mathrm{EK}} M'(\pi_1(x))$.*

The preceding two lemmas can be proved by constructing a (weak) bisimulation between the program configurations of P_0 and P'. The transition relation is $\stackrel{\mathrm{a}}{\longrightarrow}$ in both cases. A configuration of P_0 is related to a configuration of P', if

- the programs correspond to each other;
- the values and types of variables in the configuration of P_0 are related by $=_{\mathrm{EK}}$ to the first components of the values and types of variables in the configuration of P';
- the key identities and recorded plaintexts in the configuration of P' correspond to the recorded ciphertext-key-plaintext triples in the configuration of P_0.

For the concrete semantics we can show the following result.

LEMMA 5. *Let $\mathcal{O}_1$ [resp. $\mathcal{O}_2$] be the following oracle. On input of an initial memory M and the security parameter n, it executes the program P_0 [resp. P] on it (using n as the parameter for the algorithms $\mathcal{K}$, $\mathcal{E}$ and $\mathcal{D}$) and returns the public part of the final memory. Then no PPT algorithm $\mathcal{A}$ can distinguish with non-negligible advantage (in n) whether it interacts with $\mathcal{O}_1$ or $\mathcal{O}_2$.*

Indeed, plaintext integrity of the encryption system ensures that the decryption of a ciphertext with a wrong key returns $\perp$. The proof is again formalized by constructing a weak bisimulation between the program configurations of P_0 and P'. The transition relation is $\stackrel{\mathrm{c}}{\longrightarrow}$ in both cases. The bisimulation relation is similar to the proof of Lemma 4. But this time, the bisimulation is probabilistic, because the transition relation $\stackrel{\mathrm{c}}{\longrightarrow}$ is. Also, the bisimulation is with *error sets* (Backes et al. 2003) — two traces have to be similar only until one of them has reached a state from a certain *error set*. In our case the error sets correspond to situations where a ciphertext can be decrypted by a key different from the one used to create it. Such situation arises with only a negligible probability.

From a well-structured program P_0 we have now constructed a program P that is also well-structured and that is secure in the abstract or concrete setting iff the program P_0 is secure. Also, the program P contains no decryption operations. Hence it suffices to prove Theorem 2 only for programs that contain no decryptions; this, together with the given construction of the program P from the program P_0 immediately implies Theorem 2 in its full generality.

If a program P contains no decryptions then we can apply the definition of IND-CPA-security (the strongest of them in Sec. 4) of the encryption system and replace all expressions $\mathsf{enc}(k, y)$ in P with the expression $\mathsf{enc}(\mathsf{newkey}, 0)$. Let $\hat{P}$ be the resulting program. The concrete semantics of P and $\hat{P}$ are indistinguishable — a result similar to Lemma 5 can be proved for P and $\hat{P}$.

7.2 Similarity of executions

In this subsection we will show that the abstract execution of P and the concrete execution of $\hat{P}$ proceed in some sense in lock-step. Whenever we talk about the concrete semantics of the program $\hat{P}$ in this subsection, the security parameter is implicit. We are going to establish the probabilistic non-interference of $\hat{P}$ without any qualifiers about the power of the adversaries (i.e. the advantage of any adversary is the constant function 0), hence an explicit security parameter would just clutter the notation. We start by noting the following:

LEMMA 6. *Let M be an initial memory and consider an execution $C_0 \stackrel{\mathrm{a}}{\longrightarrow} C_1 \stackrel{\mathrm{a}}{\longrightarrow} \cdots \stackrel{\mathrm{a}}{\longrightarrow} C_{\mathrm{f}}$ where C_0 is the initial configuration $\langle M, \lambda x.int, \emptyset, \emptyset, P\rangle$ and C_{f} is a final configuration. Then the path of that execution through the program P (or: the values of the guard expressions at if- and while-statements) depend only on M, not on the values of keys and ciphertexts generated during the execution.*

Indeed, the guard expressions have to have the type int, but the keys and the ciphertexts have the types key and $enc(\tau)$, respectively, and there are no operations that can be applied to these values that could produce a value of type int (remember that P does not contain decryptions). Similar claims about the keys and ciphertexts not affecting the control flow of the program can be made for the concrete semantics of $\hat{P}$.

Hence for an initial memory M there exists a sequence

$$\langle \mathcal{M}_0, \Gamma_0, P_0\rangle \to \langle \mathcal{M}_1, \Gamma_1, P_1\rangle \to \cdots \to \langle \mathcal{M}_r, \Gamma_r\rangle \qquad (1)$$

where $\mathcal{M}_0 = \{M\}$, $\Gamma_0 = \lambda x.int$, $P_0 = P$, the set of memories $\mathcal{M}_i$ contains all those memories that can be reached by executing P with the initial memory M for i steps, Γ_i is the typing of variables after these i steps and P_i is the program that is still left to execute after i steps. As the path of the execution depends only on M, the typing and the remaining program are the same for all possible memories after i steps. Similarly, for the concrete semantics of $\hat{P}$ there exists a sequence

$$\langle D_0, \hat{P}_0\rangle \to \langle D_1, \hat{P}_1\rangle \to \cdots \to D_{\hat{r}} \qquad (2)$$

where D_0 is the probability distribution that puts all its weight on M, $\hat{P}_0 = \hat{P}$, D_i is the distribution over memories reached after i steps and $\hat{P}_i$ is the program that is still left to execute at this point.

We are going to show that

- $\hat{P}_i$ is the program P_i where all encryption expressions $\mathsf{enc}(k, x)$ are replaced with $\mathsf{enc}(\mathsf{newkey}, 0)$;

- $\langle \mathcal{M}_i, \Gamma_i \rangle$ uniquely determine D_i (without referring to the initial memory M)
  - the mapping from $\langle \mathcal{M}, \Gamma \rangle$ to D will be such that if the public parts of the two sets $\mathcal{M}'$, $\mathcal{M}''$ are equal then the public parts of the corresponding distributions D', D'' are also equal.

these claims imply the probabilistic non-interference of $\hat{P}$. But first we have to explore the structure of the sets $\mathcal{M}_i$ some more.

Given a typing Γ we consider the set of *extended variables* $\mathbf{EVar}_\Gamma$ that contains all "atomic" components of the variables in $\mathbf{Var}$, where "atomic" currently means "having type int, key or $enc(\tau)$". I.e. we want to refer directly to the components of the values of variables whose types are pairs. Formally, $\mathbf{EVar}_\Gamma$ is defined by the following process:

1. Let $\mathbf{EVar}_\Gamma := \mathbf{Var}$.
2. If $\mathbf{EVar}_\Gamma$ contains some z, such that $\Gamma(z) = (\tau_1, \tau_2)$ then
   - Let $\mathbf{EVar}_\Gamma := \mathbf{EVar}_\Gamma \backslash \{z\} \cup \{\pi_1(z), \pi_2(z)\}$;
   - Let $\Gamma := \Gamma[\pi_1(z) \mapsto \tau_1, \pi_2(z) \mapsto \tau_2]$
   - Go to step 2.
3. Otherwise return $\mathbf{EVar}_\Gamma$.

For some $z \in \mathbf{EVar}_\Gamma$ and a memory M whose variables are typed according to Γ we can also define the value of z in M.

LEMMA 7. *Let $\langle \mathcal{M}_i, \Gamma_i, P_i \rangle$ be an element in the sequence (1). Then the following claims hold.*

- *For all extended variables $z \in \mathbf{EVar}_{\Gamma_i}$ with $\Gamma_i(z) = int$, the value of z is the same in all $M \in \mathcal{M}_i$.*
- *There exists a partitioning Π_K of the set of all extended variables in $\mathbf{EVar}_{\Gamma_i}$ with type key, such that*
  - *for all $V \in \Pi_\mathrm{K}$, $z_1, z_2 \in V$ and $M \in \mathcal{M}_i$, $M(z_1) = M(z_2)$;*
  - *for all $V \in \Pi_\mathrm{K}$ and $M_1, M_2 \in \mathcal{M}_i$ there exists $M_3 \in \mathcal{M}_i$, such that*
    - *$M_3(z) = M_1(z)$ for all $z \in V$,*
    - *$M_3(z) =_\mathrm{E} M_2(z)$ for all $z \in \mathbf{EVar}_{\Gamma_i} \backslash V$*

    unless some key in M_2 is equal to the keys $M_1(z)$ where $z \in V$.
- *There exists a partitioning Π_E of the set of all extended variables in $\mathbf{EVar}_{\Gamma_i}$ with type $enc(\tau)$, such that*
  - *for all $V \in \Pi_\mathrm{E}$, $z_1, z_2 \in V$ and $M \in \mathcal{M}_i$, $M(z_1) = M(z_2)$;*
  - *for all $V \in \Pi_\mathrm{E}$ and $M_1, M_2 \in \mathcal{M}_i$ there exists $M_3 \in \mathcal{M}_i$, such that*
    - *$M_3(z) =_\mathrm{E} M_1(z)$ for all $z \in V$,*
    - *$M_3(z) =_\mathrm{E} M_2(z)$ for all $z \in \mathbf{EVar}_{\Gamma_i} \backslash V$*

    unless some ciphertext in M_2 is equal to the ciphertexts $M_1(z)$ where $z \in V$.

The preceding lemma states that at each step of the computation, the values that are present in the variables of the program are the following:

- Integers whose values are fixed.
- A number of keys that may each occur several times. The pattern of copying the keys is fixed. Each key takes all possible values independently of everything else. For example, it cannot happen that the possible values of a key at a certain program point are only half of all possible values for keys, the other half being cut away by some branch statement.
- A number of ciphertexts that may each occur several times. The pattern of copying the ciphertexts is fixed. *The initial vector* of each ciphertext takes all possible values independently of everything else.

Hence the set $\mathcal{M}_i$ is completely determined by Γ_i, the values of extended variables of type int and the partitions Π_K and Π_E.

The lemma is proved by induction over i, considering all possible steps that a program may make (of which there are just two — assignment and branching). The induction base is $i = 0$, the set $\mathcal{M}_0$ contains just a single memory where all variables have the type int. To simplify the induction step, assume again without lessening of generality that the program contains no nested expressions and that all guard expressions are just variables (the program P constructed in Sec. 7.1 does not satisfy this, so we have to introduce temporary variables to store intermediate results). Assume that the lemma holds for $\mathcal{M}_i$. A branching step is controlled by a guard variable of type int which has the same value in all memories in $\mathcal{M}_i$, hence $\mathcal{M}_{i+1} = \mathcal{M}_i$ in this case. An assignment step $x := e$ can be decomposed into two parts — killing the current value of x and assigning a new value to x. Killing x simply removes the values of all extended variables derived from it from the memory M, thereby possible reducing some sets in the partitions Π_K and Π_E. The effects of assigning a new value to x depend on e:

- if some values are just copied around (e is a variable, a pair or a projection) then new extended variables of type int will contain a value that is same in all memories, and the equivalence classes of Π_K and Π_E may be extended with new extended variables;
- if e is $o(x_1, \ldots, x_k)$ then the types of the variables $x_1, \ldots, x_k$ is int, their values are constant across $\mathcal{M}_i$, the operation o is deterministic, and its result is also constant;
- if e is a key generation [resp. encryption] then $\{x\}$ will be a new equivalence class in Π_K [resp. Π_E]; the possible values of x are all possible keys [resp. have all possible initial vectors].

Recall that P does not contain decryption operations.

We can now define the probability distribution $\mathbf{D}[\mathcal{M}, \Gamma]$ corresponding to a set of memories and a typing satisfying the conditions of Lemma 7 (hence the partitions Π_K and Π_E are defined). We believe that it is best described informally, by stating how a memory M is constructed when the distribution $\mathbf{D}[\mathcal{M}, \Gamma]$ is sampled. The values of the variables in M have the structure given by Γ — there is the same set of extended variables as in the memories in $\mathcal{M}$. The extended variables of type int have the same values in M as they have in $\mathcal{M}$ — these variables were constants in $\mathcal{M}$ and they are constants in $\mathbf{D}[\mathcal{M}, \Gamma]$. For each equivalence class $V \in \Pi_\mathrm{K}$ we generate a key k_V using the algorithm $\mathcal{K}$ and assign the result to all extended variables in V. For each equivalence class $V \in \Pi_\mathrm{E}$ we generate a new key k using the algorithm $\mathcal{K}$ and then encrypt the constant 0 with the key k using the algorithm $\mathcal{E}$; the resulting value is assigned to all extended variables in V. Hence the distribution $\mathbf{D}[\mathcal{M}, \Gamma]$ is the "Cartesian product" of a one-point distribution (assigning the values to extended variables of type int), a number of distributions $\mathcal{K}()$ and a number of distributions $\mathcal{E}(\mathcal{K}(), 0)$.

LEMMA 8. *Let $\mathcal{M}_i$, Γ_i and D_i be defined as in (1) and (2). Then $D_i = \mathbf{D}[\mathcal{M}_i, \Gamma_i]$.*

This lemma is again proved by induction over i, considering the possible computation steps. We omit the proof here.

We have shown that if the final sets of memories are the same for the abstract executions from initial memories M_1 and M_2 then the final distributions over memories for the concrete executions from M_1 and M_2 are the same as well. It remains to note that if just the public parts of the final sets of the memories are the same then the public parts of the final distributions are the same as well.

The public part of a set of memories $\mathcal{M}$ (together with a typing Γ, such that the conditions of Lemma 7 are satisfied) consists of

- the values of all those extended variables of type int that are parts of the variables in $\mathbf{Var}_{\mathrm{P}}$;
- the initial vectors of the ciphertexts that are the values of all those extended variables of type $enc(\tau)$ that are parts of the variables in $\mathbf{Var}_{\mathrm{P}}$.

The public part of a distribution D over memories is sampled simply by sampling D and only taking the values of variables in $\mathbf{Var}_{\mathrm{P}}$ in the resulting distribution.

The abstract security definition states precisely that modifying only the secret inputs of an initial memory does not change the public part of the resulting final set of memories. Computational non-interference (our security definition in the concrete setting) is a relaxation of probabilistic non-interference stating that the public part of the final distribution may not change if only the secret inputs of the initial memory change. Hence the abstract security of P implies the concrete security of $\hat{P}$ that is equivalent to the concrete security of P.

8. A new model

In this section we give a model that is simpler but equivalent (and sometimes even more permissive) than the semantics and the security definition in the framework for cryptographically masked flows, together with the constraints we gave in Sec. 6. In contrast to the presented abstract model, we can also publish the final values of keys in the new model — there are no constraints of $\mathbf{Var}_{\mathrm{P}}$, except that it must be disjoint with $\mathbf{Var}_{\mathrm{S}}$. Also, the program semantics in the new model is deterministic, in this aspect it is simpler than the model of cryptographically masked flows. The main components of the new model, particularly the equivalence of abstract memories, are similar to the way the equivalence is defined for formal messages by Abadi and Rogaway (2000). It is even more similar to the subsequent development of these results by Abadi and Jürjens (2001) who were also one of the first to use formal randomness to distinguish between different encryptions of the same message with the same key in the abstractions of cryptography (another early paper was (Bodei et al. 2001), but both were influenced by the idea of *confounders* by Abadi (1999)).

We have to redefine the set of values. A value $v \in \mathbf{Val}$ is either $\perp$ or defined by the following grammar:

$$v ::= b \mid \mathsf{k}\langle i\rangle \mid \{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle} \mid (v_1, v_2)$$

where $b \in \{0,1\}^*$ and $i, j \in \mathbb{N}$. The value $\mathsf{k}\langle i\rangle$ denotes the (formal) key that is produced by the i-th invocation of newkey. Similarly, $\mathsf{r}\langle j\rangle$ is the formal randomness (or: initial vector) of the j-th ciphertext.

The semantices of operations are functions from tuples of values to values. In particular, the semantics of a normal operation o of arity k is still given by a function from $(\{0,1\}^*)^k$ to $\{0,1\}^*$. If one of the arguments is not a bit-string, the result is $\perp$. The pairing takes two values v_1 and v_2 and returns (v_1, v_2), unless some of v_1, v_2 is $\perp$, in which case it returns $\perp$, too. The projections take the first or second component of a pair; if the argument of a projection is not a pair then the result is $\perp$.

The evaluation context for expressions has to contain the number n_{k} of already generated keys and the number n_{e} of already generated ciphertexts; the values of n_{k} and n_{e} are also part of program configurations. The key generation operation newkey returns $\mathsf{k}\langle n_{\mathrm{k}}+1\rangle$ and increments n_{k}. The encryption operation $\mathsf{enc}(k, y)$ expects the value of k be $\mathsf{k}\langle i\rangle$ for some $i \in \mathbb{N}$. It returns $\{v_y\}^{\mathsf{r}\langle n_{\mathrm{e}}+1\rangle}_{\mathsf{k}\langle i\rangle}$, where v_y is the value of y, and increments n_{e}. If the value of k is not a formal key or if the value of y is $\perp$, it returns $\perp$. The decryption operation expects two arguments of the form $\mathsf{k}\langle i\rangle$ and $\{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle}$ and returns v.

Again we require that in the initial memory all values are bit-strings. We define the program P to have secure information flow in our new model if for all memories M_1, M_2, M_1', M_2' where M_i' is the final memory corresponding to the initial memory M_i, $M_1 \sim_{\mathrm{P}} M_2$ implies $M_1' \cong M_2'$. The formal equivalence $\cong$ of memories is defined in the same way as by Abadi and Rogaway (2000). Formally, let $visibles(M) \subset \mathbf{Val}$ be the least set such that

- if $x \in \mathbf{Var}_{\mathrm{P}}$ then $M(x) \in visibles(M)$;
- if $(v_1, v_2) \in visibles(M)$ then $v_i \in visibles(M)$;
- if $\mathsf{k}\langle i\rangle \in visibles(M)$ and $\{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle} \in visibles(M)$ then $v \in visibles(M)$.

Let $keys(M) = visibles(M) \cap \{\mathsf{k}\langle i\rangle \mid i \in \mathbb{N}\}$. Extend the set of values by

$$v ::= \ldots \mid \square^{\mathsf{r}\langle j\rangle},$$

the value $\square^{\mathsf{r}\langle j\rangle}$ denotes a ciphertext generated using the formal coins $\mathsf{r}\langle j\rangle$, that the adversary is unable to decrypt. Let the *pattern* $pat(v, \mathbf{K})$ of a value v with respect to a set of keys $\mathbf{K}$ be defined as follows:

$$\begin{aligned}
pat(\perp, \mathbf{K}) &= \perp \\
pat(b, \mathbf{K}) &= b \\
pat(\mathsf{k}\langle i\rangle, \mathbf{K}) &= \mathsf{k}\langle i\rangle \\
pat((v_1, v_2), \mathbf{K}) &= (pat(v_1, \mathbf{K}), pat(v_2, \mathbf{K})) \\
pat(\square^{\mathsf{r}\langle j\rangle}, \mathbf{K}) &= \square^{\mathsf{r}\langle j\rangle} \\
pat(\{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle}, \mathbf{K}) &= \begin{cases} \{pat(v, \mathbf{K})\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle}, & \text{if } \mathsf{k}\langle i\rangle \in \mathbf{K} \\ \square^{\mathsf{r}\langle j\rangle}, & \text{if } \mathsf{k}\langle i\rangle \notin \mathbf{K} \end{cases}
\end{aligned}$$

Finally define $pattern(M) : \mathbf{Var}_{\mathrm{P}} \to \mathbf{Val}$ by

$$pattern(M)(x) := pat(M(x), keys(M)) \ .$$

We define two memories M_1 and M_2 as *equivalent* if $pattern(M_1)$ and $pattern(M_2)$ are α-conversions of each other. I.e. there must exist permutations $\varphi, \psi : \mathbb{N} \to \mathbb{N}$, such that if we replace each $\mathsf{k}\langle i\rangle$ in $pattern(M_1)$ with $\mathsf{k}\langle\varphi(i)\rangle$ and each $\mathsf{r}\langle j\rangle$ with $\mathsf{r}\langle\psi(j)\rangle$, we get $pattern(M_2)$.

The proof of computational soundness of the secure information flow in the new model is similar to the proof of soundness of cryptographically masked flows. Again we start with the removal of decryption operations by recording the name (identity) of the key alongside each key we generate, and the plaintext and the identity of the key alongside each ciphertext (the change is given in Table 1). There will be no change in the visible abstract semantics.

We can now define the "computational interpretation" of an abstract memory — a mapping from abstract memories to distributions over memories. The computational interpretation is actually identical to the one proposed by Abadi and Rogaway (2000); Adão et al. (2005). We show that the computational interpretation commutes with the computation steps in the abstract and concrete semantices — that the runs in the abstract and in the concrete model proceed in lock-step. Finally we use the result by Adão et al. (2005), stating that formal equivalence of memories implies the indistinguishability of the public parts of their computational interpretations.

Let us see some examples of secure and insecure programs according to the presented definition. Consider the following program

$$\begin{aligned}
k := \mathsf{newkey};\ \mathit{if}\ h\ \ &\mathit{then}\ l_1 := \mathsf{enc}(k, a);\ l_2 := \mathsf{enc}(k, b) \\
&\mathit{else}\ l_2 := \mathsf{enc}(k, a);\ l_1 := \mathsf{enc}(k, b)
\end{aligned}$$

where $\mathbf{Var}_{\mathrm{S}} = \{h\}$ and $\mathbf{Var}_{\mathrm{P}} = \{l_1, l_2\}$. The quantities a and b are constants. Depending on the initial value of h (either 0 or 1), the final memories of this program will be

$$\{h \mapsto 1, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{b\}^{\mathsf{r}\langle 2\rangle}_{\mathsf{k}\langle 1\rangle}\}$$

and

$$\{h \mapsto 0, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{b\}^{\mathsf{r}\langle 2\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}\} \ .$$

The patterns of the public parts are

$$\{l_1 \mapsto \square^{\mathsf{r}\langle 1\rangle}, l_2 \mapsto \square^{\mathsf{r}\langle 2\rangle}\}$$

and

$$\{l_1 \mapsto \square^{\mathsf{r}\langle 2\rangle}, l_2 \mapsto \square^{\mathsf{r}\langle 1\rangle}\} \ .$$

We see that the patterns do not depend on the constants a and b. Furthermore, the first of them can be α-converted to the second one by letting the permutation ψ of formal randomness indices map 1 to 2 and 2 to 1. Hence the considered program is secure.

Consider now the following program:

$$k := \mathsf{newkey};\ l_1 := \mathsf{enc}(k, a);\ \mathit{if}\ h\ \ \mathit{then}\ l_2 := \mathsf{enc}(k, b)\ \ \mathit{else}\ l_2 := l_1$$

with the same $\mathbf{Var}_{\mathrm{S}}$, $\mathbf{Var}_{\mathrm{P}}$, a and b as before. Depending on the value of h, the final memories of this program will be

$$\{h \mapsto 1, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{b\}^{\mathsf{r}\langle 2\rangle}_{\mathsf{k}\langle 1\rangle}\}$$

and

$$\{h \mapsto 0, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}\} \ .$$

The patterns of the public parts are

$$\{l_1 \mapsto \square^{\mathsf{r}\langle 1\rangle}, l_2 \mapsto \square^{\mathsf{r}\langle 2\rangle}\}$$

and

$$\{l_1 \mapsto \square^{\mathsf{r}\langle 1\rangle}, l_2 \mapsto \square^{\mathsf{r}\langle 1\rangle}\} \ .$$

As there exists no α-conversion between these possible patterns, the program is not secure. Indeed, the value of h can be determined by considering the equality of l_1 and l_2. This example (called the *occlusion example*) was presented by Askarov et al. (2006) as one of the motivating examples for the equivalence relation $=_{\mathrm{E}}$ among ciphertexts.

9. Related work

Cryptographically masked flows were proposed by Askarov et al. (2006) along with a type system for checking the security of information flow. Askarov and Sabelfeld (2007) considered the release of keys in this framework, using it to provide a mechanism for declassification.

In quite general terms, the topic of this paper is secure information flow. A semi-recent overview of this topic is given by Sabelfeld and Myers (2003). The handling of encryption and publishing the ciphertexts is also closely connected to the topic of declassification, a good overview of which is given by Sabelfeld and Sands (2005).

In this paper we have searched for *abstractions* of cryptography; we have been particularly concerned with the soundness of such abstractions. The most well-known and celebrated abstraction of cryptography is without doubt the Dolev-Yao model (Dolev and Yao 1983) that abstracts the cryptographic messages by terms in a free algebra and lists all possible means to derive new messages from the known ones. The Dolev-Yao model even appears in this paper, in Sec. 8. The question on the soundness has been unanswered for a long time, the first well-known results connecting it to the computational model were given by Abadi and Rogaway (2000). Abadi and Rogaway (2000) considered only passive adversaries, but the soundness (for integrity properties) in the presence of active adversaries was shown by Cortier and Warinschi (2005). Another, earlier soundness proof was given by Herzog (2002), but this proof required plaintext aware encryption systems for which no constructions without random oracles are known. Also, a comprehensive soundness result was established by the presentation of the universally composable cryptographic library (Backes et al. 2003) showing that minor adjustments to the Dolev-Yao model indeed cause it to reflect all observable properties of the computational model. Still, those results about the Dolev-Yao model do not carry that easily over to cryptographically masked flows.

Recently, some other abstractions of the encryption functionality have appeared, although the focus of these works has been different — an information-hiding and integrity-preserving construction has been proposed and then it is shown how to implement it using cryptographic primitives. Vaughan and Zdancewic (2007) have proposed an *information-packing* primitive that declassifies its argument from its original security level to the lowest level, but at the same time preserves its confidentiality and integrity as if it had not been declassified. The packing primitive is integrated into the decentralized label model (Myers 1999) and implemented using public-key encryption and signatures. Fournet and Rezk (2008) give a sound implementation of the shared memory security model (stating which principal is allowed to read or write which variables) using cryptography (again, public-key encryption and signatures). The soundness of the implementation is shown using language-based techniques, by giving type systems for secure information flow for both the source and target languages (which may be of independent interest) and showing that the translation preserves typing.

10. Discussion

This paper has demonstrated the limits of cryptographically masked flows. Some of them are quite natural (e.g. the handling of keys), but one of them in particular may seem excessive, partly because the program analyses working directly on the computational semantics (Laud 2001, 2003; Laud and Vene 2005) do not have that limit. We are concerned about the inability to use the "actual values" of ciphertexts in arbitrary computations.

Still, as Sec. 5.4 demonstrated, we likely cannot allow the arbitrary handling of random values (in the concrete semantics), unless these values are somehow randomly generated in the abstract semantics as well. Models where the encryption is replaced with random number generation have been considered (Smith and Alpízar 2006) and they would be quite similar to the programs that we get after performing the replacement of the real functionality of the encryption primitive with the ideal functionality. If we also want to have decryption operations in such a model then we have to keep a table of plaintext-key-ciphertext pairs that have occurred in the course of the computation, similarly to the modifications of the abstract semantics in Sec. 5.5. The practical value of such abstract model is strongly dependent on the available tool support. Fortunately, tools for arguing about stochastic programs will suffice; the tools do not have to deal with the cryptographic effects because these have been abstracted away.

11. Acknowledgements

This research has been supported by Estonian Science Foundation, grant No. 6944 and by EU Integrated Project MOBIUS (contract no. IST-15905). We thank the anonymous referees and Steve Zdancewic for their valuable comments, as well as Pierpaolo Degano for pointing out the origins of formal randomness.

References

Martín Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

Martín Abadi and Jan Jürjens. Formal Eavesdropping and Its Computational Interpretation. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, volume 2215 of *LNCS*, pages 82–94, Sendai, Japan, October 2001. Springer-Verlag.

Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 3–22, Sendai, Japan, August 2000. Springer-Verlag.

Pedro Adão, Gergei Bana, Jonathan Herzog, and Andre Scedrov. Soundness of formal encryption in the presence of key-cycles. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 374–396. Springer, 2005.

Johan Agat. Transforming out timing leaks. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, Boston, Massachusetts, January 2000. ACM Press.

Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In Pfitzmann and McDaniel (2007), pages 207–221.

Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-Masked flows. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 353–369. Springer, 2006.

Michael Backes, Birgit Pfitzmann, and Michael Waidner. A Universally Composable Cryptographic Library. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, October 2003. ACM Press. Extended version available as Report 2003/015 of Cryptology ePrint Archive.

Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security*, volume 1976 of *LNCS*, pages 531–545, Kyoto, Japan, December 2000. Springer-Verlag.

Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, November 1993. ACM Press.

Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *38th Annual Symposium on Foundations of Computer Science*, pages 394–403, Miami Beach, Florida, October 1997. IEEE Computer Society Press.

John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2002.

Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. In Victor E. Malyshkin, editor, *PaCT*, volume 2127 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2001.

Veronique Cortier and Bogdan Warinschi. Computationally Sound, Automated Proofs for Security Protocols. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *LNCS*, pages 157–171, Edinburgh, UK, April 2005. Springer-Verlag.

Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

Cédric Fournet and Tamara Rezk. Cryptographically Sound Implementations for Typed Information-Flow Security. In *POPL 2008, Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 2008. ACM Press.

Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, April 1982. IEEE Computer Society Press.

Oded Goldreich. *Foundations of Cryptography. Volume 1 - Basic Tools*. Cambridge University Press, 2001.

Jonathan Herzog. Computational Soundness of Formal Adversaries. Master's thesis, Massachusetts Institute of Technology, September 2002.

Peeter Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *LNCS*, pages 77–91, Genova, Italy, April 2001. Springer-Verlag.

Peeter Laud. Handling Encryption in Analyses for Secure Information Flow. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, pages 159–173, Warsaw, Poland, April 2003. Springer-Verlag.

Peeter Laud and Varmo Vene. A Type System for Computationally Secure Information Flow. In Maciej Liśkiewicz and Rüdiger Reischuk, editors, *15th International Symposium on Fundamentals of Computation Theory (FCT) 2005*, volume 3623 of *LNCS*, pages 365–377, Lübeck, Germany, August 2005. Springer-Verlag.

John McLean. Security models and information flow. In *IEEE Symposium on Security and Privacy*, pages 180–189, 1990.

Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press.

Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

Birgit Pfitzmann and Patrick McDaniel, editors. *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, 2007. IEEE Computer Society.

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

Andrei Sabelfeld and David Sands. A Per Model of Secure Information Flow in Sequential Programs. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 40–58, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW*, pages 255–269. IEEE Computer Society, 2005.

Geoffrey Smith and Rafael Alpízar. Secure Information Flow with Random Assignment and Encryption. In *4th ACM Workshop on Formal Methods in Security Engineering*, pages 33–43, 2006.

Geoffrey Smith and Dennis M. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998. ACM Press.

Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In Pfitzmann and McDaniel (2007), pages 192–206.