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

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 $\mathtt{entry} = 1$. 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`).

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

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:*

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

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:*

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

we transform the program into:

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

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

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

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

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

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

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

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

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.