Integration of a Security Type System into a Program Logic*

Reiner Hähnle[1], Jing Pan[2], Philipp Rümmer[1], and Dennis Walter[1]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
[2] Department of Mathematics and Computer Science,
Eindhoven University of Technology

Abstract. Type systems and program logics are often conceived to be at opposing ends of the spectrum of formal software analyses. In this paper we show that a flow-sensitive type system ensuring non-interference in a simple while language can be expressed through specialised rules of a program logic. In our framework, the structure of non-interference proofs resembles the corresponding derivations in a recent security type system, meaning that the algorithmic version of the type system can be used as a proof procedure for the logic. We argue that this is important for obtaining uniform proof certificates in a proof-carrying code framework. We discuss in which cases the interleaving of approximative and precise reasoning allows us to deal with delimited information release. Finally, we present ideas on how our results can be extended to encompass features of realistic programming languages like Java.

1 Introduction

Formal verification of software properties has recently attracted a lot of interest. An important factor in this trend is the enormously increased need for secure applications, particularly in mobile environments. Confidentiality policies can often be expressed in terms of information flow properties. Existing approaches to verification of such properties mainly fall into two categories: the first are type-based security analyses ([20] gives an overview), whereas the second are deduction-based employing program logics (e.g. [13, 5, 9]).

It is often noted that type-based analyses have a very logic-like character: A language for judgements is provided, a semantics that determines the set of *valid* judgments, and finally type rules to approximate the semantics mechanically. Type systems typically can trade a precise reflection of the semantics of judgments for automation and efficiency: many valid judgments are rejected.

* This work was funded in part by a STINT institutional grant and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

For program logics, the situation is quite the opposite: Calculi try to capture the semantics as precisely as possible and therefore have significantly higher complexity than type systems. Furthermore, due to the richer syntax of program logics – compared to the judgments in the type world – the framework is more general and the same program logic can be used to express and reason about different kinds of program properties.

The main contributions of this paper are: we construct a calculus for a program logic that naturally simulates the rules of a flow-sensitive type system for secure information flow. We prove soundness of the program logic calculus with respect to the type system. The so obtained interpretation of the type system in dynamic logic yields increased precision and opens up ways of expressing properties beyond pure non-interference. Concretely, we are able to prove the absence of exceptions in certain cases, and we can express delimited information release. Therefore, we can speak of an integration of a security type system into program logic.

A crucial benefit of the integration is that we obtain an automatic proof procedure for non-interference formulae: because of the similarity between the program logic calculus and the type rules, it is possible to mechanically translate type derivations to deduction proofs in the program logic. At the same time, certain advantages over the type system in terms of precision (Sect. 5) come for free without sacrificing automation.

The paper is organised as follows. In Section 2 we argue that a formal connection between type systems and program logics fits nicely into a verification strategy for advanced security policies of mobile JAVA programs based on proof-carrying-code (PCC). Section 3 introduces the terminology used in the rest of the paper. In Section 4 we define and discuss our program logic tailored to non-interference analysis. Our ideas for increasing the precision of the calculus and for covering delimited information release are given in Section 5. Due to lack of space, we could not include proofs in this paper. An extended version with all proofs is provided at [10].

2 Integrating Type Systems and Program Logics

We think that the integration of type systems and program logics is an important ingredient to make security policy checks scale up to mobile code written in modern industrial programming languages.

Certificates for Proof-Carrying Code. For the security infrastructure of mobile, ubiquituous computing it is essential that security policies can be enforced locally on the end-user device without requiring a secure internet connection to a trusted authentication authority. In the EU project MOBIUS[3] this infrastructure is based on the proof-carrying code (PCC) technology [16]. The basic idea of PCC is to provide a formal proof that a security policy holds for a given application, and then to hand down to the code consumer (end user) not only the

[3] `mobius.inria.fr/twiki/bin/view/Mobius`

application code, but also a certificate that allows to reconstruct the security proof locally with low overhead. Therefore, the end user device must run a proof checker, and, in a standard PCC architecture [16], also a verification condition generator, because certificates do not contain aspects of programs. The latter makes the approach unpractical for devices with limited resources. In addition, the security policies considered in Mobius [14] are substantially more complex than the safety policies originally envisioned in PCC. In foundational PCC [4] this is dispensed with at the price of including the formal semantics of the target language in the proof checker. The size of the resulting proof certificates makes this approach impractical so far. In the case of an axiomatic semantics as used in the verification system employed in the present paper [1], it seems possible to arrive at a *trusted code base* that is small enough. In the type-based version of PCC the trusted code base consists of a type checker instead of a proof checker. The integration of a type system for secure information flow into a program logic makes it possible to construct uniformly logic-based certificates, and no hybrid certificates need to be maintained. As a consequence, the PCC architecture is simplified and the trusted code base is significantly reduced. Efforts that go into similar directions in the sense that the scope of certificates is extended include Configurable PCC [17] and Temporal Logic PCC [8].

Synergies from Combining Type-Based and Deduction-Based Verification. The possibility to combine type-based and deduction-based reasoning in one framework leads to a number of synergies. In an integrated type- and deduction-based framework it is possible to increase the precision of the analysis dynamically on demand. Type systems ignore the values of variables. In a deduction framework, however, one can, e.g., prove that in the program "$\textbf{if}\ (b)\ y = x\ ; \textbf{if}\ (\neg b)\ z = y\ ;$" the variables z and x are independent, because the value of b always excludes the path through one of the conditionals. Note that it is not necessary to track the values of all variables to determine this: only the value of b matters in the example. More realistic examples are in Sect. 5.

A further opportunity offered by the integration of type-based analysis into an expressive logical framework is the formulation of additional security properties without the need for substantial changes in the underlying rule system or the deduction engine. To illustrate this point we show in Sect. 5 that it is possible to express delimited information release in our program logic.

3 Background and Terminology

3.1 Non-Interference Analysis

Generally speaking, a program has secure information flow if no knowledge about some given secret data can be gained by executing this program. Whether or not a program has secure information flow can hence only be decided according to a given security policy discriminating secret from public data. In our considerations we adopt the common model where all input and output channels are taken to be program variables. The semantic concept underlying secure information flow then is that of non-interference: nothing can be learned about a secret initially stored in variable `h`, by observing variable `l` after program execution, if the initial value of `h` *does not interfere with* the final value of `l`. Put differently, the final value of `l` must be *independent* of the initial value of `h`.

This non-interference property is commonly established via security type systems [20, 12, 21, 2], where a program is deemed secure if it is typable according to some given policy. Type systems are used to perform flow-sensitive as well as flow-insensitive analyses. Flow-insensitive approaches (e.g. [21]) require every subprogram to be well-typed according to the *same* policy. Recent flow-sensitive analyses [12, 2] allow the types of variables to change along the execution path, thereby providing more flexibility for the programmer. Like these type systems, the program logic developed in this paper will be termination insensitive, meaning that a security guarantee is only made about terminating runs of the program under consideration.

The type system of Hunt & Sands [12] is depicted in Fig. 1. The type p represents the security level of the program counter and serves to eliminate indirect information flow. The remaining components of typing judgments are a program α and two typing functions $\nabla, \nabla' : \mathrm{PVar} \to \mathcal{L}$ mapping program variables to their respective pre- and post-types. The type system is parametric with respect to the choice of security types; it only requires them to form a (complete) lattice $\mathcal{L}$. In this paper, we will only consider the most general[4] lattice $\mathcal{P}(\mathrm{PVar})$. One may thus think of the type $\nabla(v)$ of a variable v as the set of all variables that v's value may depend on at a given point in the program. A judgment $p \vdash^{\mathrm{HS}} \nabla \ \{ \ \alpha \ \} \ \nabla'$ states that in context p the program α transforms the typing (or dependency approximation) ∇ into ∇'. We note that rule Assign$^{\mathrm{HS}}$ gives the system its flow-sensitive character, stating that variable v's type is changed by an assignment $v = E$ to E's type as given by the pre-typing ∇ joined with the context type p. The type t of an expression E in a typing ∇ can simply be taken to be the join of the types $\nabla(v)$ of all free variables v occurring in E, which we denote by $\nabla \vdash E : t$. Joining with the context p is required to accomodate for leakage through the program context, as in the program "**if** (h) $\{l = 1\}$ $\{l = 0)\}$", where the initial value of h is revealed in the final value of l. A modification of the context p can be observed, e.g., in rule If$^{\mathrm{HS}}$, where the subderivation of the two branches of an if statement must be conducted in a context lifted by the type of the conditional.

3.2 Dynamic Logic with Updates

Following [9], the program logic that we investigate is a simplified version of dynamic logic (DL) for JavaCard [6]. The most notable difference to standard first-order dynamic logic for the simple while-language [11] is the presence of an explicit operator for simultaneous substitutions (called *updates* [19]). While updates become particularly useful when more complicated programming lan-

[4] In the sense that any other type lattice is subsumed by it, see [12, Lem. 6.8].

$$\frac{}{p \vdash^{\mathrm{HS}} \nabla \; \{ \; \} \; \nabla} \; \textsc{Skip}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash E : t}{p \vdash^{\mathrm{HS}} \nabla \; \{ \; v = E \; \} \; \nabla[v \mapsto p \sqcup t]} \; \textsc{Assign}^{\mathrm{HS}}$$

$$\frac{p \vdash^{\mathrm{HS}} \nabla \; \{ \; \alpha_1 \; \} \; \nabla' \qquad p \vdash^{\mathrm{HS}} \nabla' \; \{ \; \alpha_2 \; \} \; \nabla''}{p \vdash^{\mathrm{HS}} \nabla \; \{ \; \alpha_1 \; ; \; \alpha_2 \; \} \; \nabla''} \; \textsc{Seq}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash b : t \qquad p \sqcup t \vdash^{\mathrm{HS}} \nabla \; \{ \; \alpha_i \; \} \; \nabla' \;\; (i = 1, 2)}{p \vdash^{\mathrm{HS}} \nabla \; \{ \; \mathbf{if} \; b \; \alpha_1 \; \alpha_2 \; \} \; \nabla'} \; \textsc{If}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash b : t \qquad p \sqcup t \vdash^{\mathrm{HS}} \nabla \; \{ \; \alpha \; \} \; \nabla}{p \vdash^{\mathrm{HS}} \nabla \; \{ \; \mathbf{while} \; b \; \alpha \; \} \; \nabla} \; \textsc{While}^{\mathrm{HS}}$$

$$\frac{p_1 \vdash^{\mathrm{HS}} \nabla_1 \; \{ \; \alpha \; \} \; \nabla_1'}{p_2 \vdash^{\mathrm{HS}} \nabla_2 \; \{ \; \alpha \; \} \; \nabla_2'} \; \textsc{Sub}^{\mathrm{HS}} \qquad p_2 \sqsubseteq p_1, \nabla_2 \sqsubseteq \nabla_1, \; \nabla_1' \sqsubseteq \nabla_2'$$

Fig. 1. Hunt & Sands' flow-sensitive type system for information flow analysis

guages (with arrays or object-oriented features) are considered, in any case, they enable a more direct relation between program logic and type systems.

A *signature* of DL is a tuple $(\Sigma, \mathrm{PVar}, \mathrm{LVar})$ consisting of a set Σ of *function symbols* with fixed, non-negative arity, a set PVar of *program variables* and of a countably infinite set LVar of *logical variables.* Σ, PVar, LVar are pairwise disjoint. Because some of our rules need to introduce fresh function symbols, we assume that Σ contains infinitely many symbols for each arity n. Further, we require that a distinguished nullary symbol $\mathit{TRUE} \in \Sigma$ exists. *Rigid terms* t_{r}, *ground terms* t_{g}, *terms* t,[5] *programs* α, *updates* U and *formulae* ϕ are then defined by the following grammar, where $f \in \Sigma$ ranges over functions, $x \in \mathrm{LVar}$ over logical variables and $v \in \mathrm{PVar}$ over program variables:

$$\begin{array}{rcl}
t_{\mathrm{r}} & ::= & x \mid f(t_{\mathrm{r}}, \ldots, t_{\mathrm{r}}) \\
t & ::= & t_{\mathrm{r}} \mid t_{\mathrm{g}} \mid f(t, \ldots, t) \mid \{U\}\, t \\
\phi & ::= & \phi \wedge \phi \mid \forall x.\ \phi \mid \ \ldots \ \mid t = t \mid [\alpha]\, \phi \mid \{U\}\, \phi \\
\alpha & ::= & \alpha \; ; \; \ldots \; ; \; \alpha \mid v = t_{\mathrm{g}} \mid \mathbf{if}\; t_{\mathrm{g}}\; \alpha\; \alpha \mid \mathbf{while}\; t_{\mathrm{g}}\; \alpha
\end{array}
\qquad
\begin{array}{rcl}
t_{\mathrm{g}} & ::= & v \mid f(t_{\mathrm{g}}, \ldots, t_{\mathrm{g}}) \\
U & ::= & \epsilon \mid v := t,\ U
\end{array}$$

For the whole paper, we assume a fixed signature $(\Sigma, \mathrm{PVar}, \mathrm{LVar})$ in which the set $\mathrm{PVar} = \{v_1, \ldots, v_n\}$ is finite, containing exactly those variables occurring in the progam under investigation.

A *structure* is a pair $S = (D, I)$ consisting of a non-empty *universe* D and an *interpretation* I of function symbols, where $I(f) : D^n \to D$ if $f \in \Sigma$ has arity n. *Program variable assignments* and *variable assignments* are mappings

[5] Both rigid terms and ground terms are terms.

$\delta : \mathrm{PVar} \to D$ and $\beta : \mathrm{LVar} \to D$. The space of all program variable assignments over the universe D is denoted by $PA^D = \mathrm{PVar} \to D$, and the corresponding flat domain by $PA^D_\perp = PA^D \cup \{\perp\}$, where $\delta \sqsubseteq \delta'$ iff $\delta = \perp$ or $\delta = \delta'$.

While-programs α are evaluated in structures and operate on program variable assignments. We use a standard denotational semantics for such programs

$$[\![\alpha]\!]^S : PA^D \to PA^D_\perp$$

and define, for instance, the meaning of a loop "**while** b α" through

$$[\![\mathbf{while}\ b\ \alpha]\!]^S =_{\mathrm{def}} \bigsqcup_i w_i, \qquad w_i : PA^D \to PA^D_\perp$$

$$w_0(\delta) =_{\mathrm{def}} \perp, \quad w_{i+1}(\delta) =_{\mathrm{def}} \begin{cases} (w_i)_\perp([\![\alpha]\!]^S(\delta)) & \text{for } val_{S,\delta}(b) = val_S(\mathit{TRUE}) \\ \delta & \text{otherwise} \end{cases}$$

where we make use of a 'bottom lifting': $(f)_\perp(x) = \mathit{if}\ (x = \perp)\ \mathit{then} \perp \mathit{else}\ f(x)$.

Likewise, updates are given a denotation as total operations on program variable assignments. The statements of an update are executed in parallel and statements that literally occur later can override the effects of earlier statements:

$$\begin{aligned} &[\![U]\!]^{S,\beta} : PA^D \to PA^D \\ &[\![w_1 := t_1, \ldots, w_k := t_k]\!]^{S,\beta}(\delta) =_{\mathrm{def}} \\ &\quad (\cdots((\delta[w_1 \mapsto val_{S,\beta,\delta}(t_1)])[w_2 \mapsto val_{S,\beta,\delta}(t_2)])\cdots)[w_k \mapsto val_{S,\beta,\delta}(t_k)] \end{aligned}$$

where $(\delta[w \mapsto a])(v) = \mathit{if}\ (v = w)\ \mathit{then}\ a\ \mathit{else}\ \delta(v)$ are ordinary function updates.

Evaluation $val_{S,\beta,\delta}$ of terms and formulae is mostly defined as it is common for first-order predicate logic. Formulas are mapped into a Boolean domain, where tt stands for semantic truth. The cases for programs and updates are

$$\begin{aligned} val_{S,\beta,\delta}([\alpha]\,\phi) &=_{\mathrm{def}} \begin{cases} val_{S,\beta,[\![\alpha]\!]^S(\delta)}(\phi) & \text{for } [\![\alpha]\!]^S(\delta) \neq \perp \\ \mathrm{tt} & \text{otherwise} \end{cases} \\ val_{S,\beta,\delta}(\{\,U\,\}\,\phi) &=_{\mathrm{def}} val_{S,\beta,[\![U]\!]^{S,\beta}(\delta)}(\phi) \end{aligned}$$

We interpret free logical variables $x \in \mathrm{LVar}$ existentially: a formula ϕ is *valid* iff for each structure $S = (D, I)$ and each program variable assignment $\delta \in PA^D$ there is a variable assignment $\beta : \mathrm{LVar} \to D$ such that $val_{S,\beta,\delta}(\phi) = \mathrm{tt}$. Likewise, a sequent $\Gamma \vdash^{\mathrm{dl}} \Delta$ is called valid iff $\bigwedge \Gamma \to \bigvee \Delta$ is valid.

The set of unbound variables occurring in a term or a formula t is denoted by $\mathrm{vars}(t) \subseteq \mathrm{PVar} \cup \mathrm{LVar}$. For program variables $v \in \mathrm{PVar}$, this means $v \in \mathrm{vars}(t)$ iff v turns up anywhere in t. For logical variables $x \in \mathrm{LVar}$, we define $x \in \mathrm{vars}(t)$ iff x occurs in t and is not in the scope of $\forall x$ or $\exists x$.

We note that the semantic notion of non-interference can easily be expressed in the formalism of dynamic logic: One possibility [9] is to express the variable independence property introduced above as follows. Assuming the set of program variables is $\mathrm{PVar} = \{v_1, \ldots, v_n\}$, then v_j only depends on $v_1, \ldots, v_i$ if variation of $v_{i+1}, \ldots, v_n$ does not affect the final value of v_j:

$$\forall u_1, \ldots, u_i.\ \exists r.\ \forall u_{i+1}, \ldots, u_n.\ \{\, v_i := u_i \,\}_{1 \leq i \leq n}\ [\alpha]\,(v_j = r) \ . \qquad (1)$$

The particular use of updates in this formula is a standard trick to quantify over program variables which is not allowed directly: in order to quantify over all values that a program variable v occurring in a formula ϕ can assume, we introduce a fresh logical variable u and quantify over the latter. In the following we use quantification over program variables as a shorthand, writing $\dot{\forall} v.\ \phi$ for $\forall u.\ \{\, v := u \,\}\ \phi$. One result of this paper is that simple, easily automated proofs of formulae such as (1) are viable in at least those cases where a corresponding derivation in the type system of Hunt and Sands exists.

4 Interpreting the Type System in Dynamic Logic

We now present a calculus for dynamic logic in which the rules involving program statements employ abstraction instead of precise evaluation. The calculus facilitates automatic proofs of secure information flow. In particular, when proving loops the burden of finding invariants is reduced to the task of providing a dependency approximation between program variables. There is a close correspondence to the type system of [12] (Fig. 1). Intuitively, state updates in the DL calculus resemble security typings in the type system: updates arising during a proof will essentially take the form $\{\, v := f(\ldots \mathit{vars} \ldots) \,\}$, where the *vars* form the type of v in a corresponding derivation in the type system. To put our observation on a formal basis, we prove the soundness of the calculus and show that every derivation in the type system has a corresponding proof in our calculus.

The Abstraction-based Calculus. We introduce *extended type environments* as pairs (∇, I) consisting of a typing function $\nabla : \mathrm{PVar} \to \mathcal{P}(\mathrm{PVar})$ and an *invariance set* $I \subseteq \mathrm{PVar}$ used to indicate those variables whose value does not change after execution of the program. We write ∇_v for the syntactic sequence of variables $w_1, \ldots, w_k$ with arbitrary ordering when $\nabla(v) = \{w_1, \ldots, w_k\}$ and ∇_v^C for a sequence of all variables *not* in $\nabla(v)$. Ultimately, we want to prove non-interference properties of the form

$$\{\ \alpha\ \} \Downarrow (\nabla, I) \quad \equiv_{\mathrm{def}} \quad \bigwedge_{v \in \mathrm{PVar}} \begin{cases} \dot{\forall} v_1 \cdots v_n.\ \forall u.\ \{\, v := u \,\}[\alpha]\, v = u & , v \in I \\ \dot{\forall} \nabla_v.\ \exists r.\ \dot{\forall} \nabla_v^C.\ [\alpha]\, v = r & , v \notin I \end{cases} \tag{2}$$

where we assume $\mathrm{PVar} = \{v_1, \ldots, v_n\}$. Validity of a judgment $\{\ \alpha\ \} \Downarrow (\nabla, I)$ ensures that all variables in the invariance set I remain unchanged after execution of the program α, and that any variable v of the rest only depends on variables in $\nabla(v)$. The invariance set I corresponds to the context p that turns up in judgments $p \vdash^{\mathrm{HS}} \nabla\ \{\ \alpha\ \}\ \nabla'$: while the type system ensures that p is a lower bound of the post-type $\nabla'(v)$ of variables v assigned in α, the set I can be used to ensure that variables with low post-type are not assigned (or, more precisely, not changed). The equivalence is formally stated in Lem. 2.

In the proof process we want to abstract program statements "**while** b α" and "**if** b α_1 α_2" into updates modelling the effects of these statements. Thus we avoid having to split up the proof for the two branches of an if-statement, or having to find an invariant for a while-loop. Extended type environments capture the essence of these updates: the arguments for the abstraction functions and the unmodified variables. They are translated into updates as follows:

$$\begin{aligned}\mathrm{upd}(\nabla, I) \quad &=_{\mathrm{def}} \quad \{\, v := f_v(\nabla_v)\,\}_{v \in \mathrm{PVar} \setminus I} \\ \mathrm{ifUpd}(b, \nabla, I) \quad &=_{\mathrm{def}} \quad \{\, v := f_v(b, \nabla_v)\,\}_{v \in \mathrm{PVar} \setminus I}\end{aligned}$$

The above updates assign to each v not in the invariance set I a *fresh* function symbol f_v whose arguments are exactly the variables given by the type $\nabla(v)$. In a program "**if** $b\ \alpha_1\ \alpha_2$" the final state may depend on the branch condition b, so the translation ifUpd 'injects' the condition into the update. This is the analogon of the context lifting present in $\mathrm{IF}^{\mathrm{HS}}$. For the while-rule, we transform the loop body into a conditional, so that we must handle the context lifting only in the if-rule.

Figs. 2 and 3 contain the rules for a sequent calculus. We have only included those propositional and first-order rules (the first four rules of Fig. 2) that are necessary for proving the results in this section; more rules are required to make the calculus usable in practice. The calculus uses free logical variables $X \in \mathrm{LVar}$ ($\mathrm{EX\text{-}RIGHT}^{\mathrm{dl}}$) and unification ($\mathrm{CLOSE\text{-}EQ}^{\mathrm{dl}}$) for handling existential quantification, where the latter rule works by applying the unifier of terms s and t to the whole proof tree. We have to demand that only rigid terms (not containing program variables) are substituted for free variables, because free variables can also occur in the scope of updates or the box modal operator. Skolemisation ($\mathrm{ALL\text{-}RIGHT}^{\mathrm{dl}}$) has to collect the free variables that occur in a quantified formula to ensure soundness. By definition of the non-interference properties (2) and by the design of the rules of the dynamic logic calculus it is sufficient to define update rules for terms, quantifier-free formulae, and other updates. Such rules can be used at any point in a proof to simplify expressions containing updates.

Rule $\mathrm{ABSTRACT}^{\mathrm{dl}}$ can be used to normalise terms occuring in updates to the form $f(\ldots \mathit{vars} \ldots)$. In rules $\mathrm{IF}^{\mathrm{dl}}$ and $\mathrm{WHILE}^{\mathrm{dl}}$ the second premiss represents the actual abstraction of the program statement for a suitably chosen typing ∇ and invariance set I. This abstraction is justified through the first premiss in terms of another non-interference proof obligation. The concretisation operator γ^* (cf. [12]) of rule $\mathrm{WHILE}^{\mathrm{dl}}$ is generally defined as

$$\gamma^*_{\nabla_1}(\nabla_2)(x) \ =_{\mathrm{def}} \ \{y \in \mathrm{PVar} \mid \nabla_1(y) \subseteq \nabla_2(x)\} \qquad (x \in \mathrm{PVar}) \ . \tag{3}$$

Together with the side condition that for all v we require $v \in \nabla(v)$, a closure property on dependencies is ensured: $w \in \gamma^*_{\nabla}(\nabla)(v)$ implies $\gamma^*_{\nabla}(\nabla)(w) \subseteq \gamma^*_{\nabla}(\nabla)(v)$: if a variable depends on another, the latter's dependencies are included in the former's. This accounts for the fact that the loop body can be executed more than once, which, in general, causes transitive dependencies.

Function Arguments Ensure Soundness. A recurring proof obligation in a non-interference proof is a statement of the form $\dot\forall \nabla_v.\ \exists r.\ \dot\forall \nabla^C_v.\ [\alpha]\, v = r$. To prove

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi, \Delta \qquad \Gamma \vdash^{\mathrm{dl}} \psi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \phi \land \psi, \Delta}\ \text{AND-RIGHT}^{\mathrm{dl}}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi[x/f(X_1,\dots,X_n)], \Delta}{\Gamma \vdash^{\mathrm{dl}} \forall x.\ \phi, \Delta}\ \text{ALL-RIGHT}^{\mathrm{dl}} \qquad \{X_1,\dots,X_n\} = \mathrm{vars}(\phi) \cap \mathrm{LVar}\backslash\{x\},\ f \text{ fresh}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi[x/X], \exists x.\ \phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \exists x.\ \phi, \Delta}\ \text{EX-RIGHT}^{\mathrm{dl}} \qquad X \text{ fresh}$$

$$\frac{\overset{*}{[\, s \equiv t\,]}}{\Gamma \vdash^{\mathrm{dl}} s = t, \Delta}\ \text{CLOSE-EQ}^{\mathrm{dl}} \qquad s, t \text{ unifiable (with rigid unifier)}$$

$$\frac{(\Gamma \vdash^{\mathrm{dl}} \Delta)[x/f(\mathrm{vars}(t))]}{(\Gamma \vdash^{\mathrm{dl}} \Delta)[x/t]}\ \text{ABSTRACT}^{\mathrm{dl}} \qquad f \text{ fresh}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, \phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, [\,]\, \phi, \Delta}\ \textsc{Skip}^{\mathrm{dl}} \qquad \frac{\Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, \{\, v := E\,\}\, [\dots]\, \phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, [\, v = E\ ;\ \dots]\, \phi, \Delta}\ \textsc{Assign}^{\mathrm{dl}}$$

$$\frac{\begin{array}{c}\vdash^{\mathrm{dl}} \{\ \alpha_i\ \} \Downarrow (\nabla, I) \quad (i = 1, 2) \\ \Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, \{\, \mathrm{ifUpd}(b, \nabla, I)\,\}\, [\dots]\, \phi, \Delta\end{array}}{\Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, [\,\mathbf{if}\ b\ \alpha_1\ \alpha_2\ ;\ \dots]\, \phi, \Delta}\ \textsc{If}^{\mathrm{dl}}$$

$$\frac{\begin{array}{c}\vdash^{\mathrm{dl}} \{\ \mathbf{if}\ b\ \alpha\ \{\}\ \} \Downarrow (\gamma^*_{\nabla}(\nabla), I) \\ \Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, \{\, \mathrm{upd}(\nabla, I)\,\}\, [\dots]\, \phi, \Delta\end{array}}{\Gamma \vdash^{\mathrm{dl}} \{\, U\,\}\, [\,\mathbf{while}\ b\ \alpha\ ;\ \dots]\, \phi, \Delta}\ \textsc{While}^{\mathrm{dl}} \qquad v \in \nabla(v) \text{ for all } v \in \mathrm{PVar}$$

Fig. 2. A dynamic logic calculus for information flow security. In the last four rules the update $\{\, U\,\}$ can also be empty and disappear.

this statement without abstraction essentially is to find a function of the variables ∇_v that yields the value of v under α for every given pre-state: one must find the strongest post-condition w.r.t. v's value. Logically, one must create this function as a term for the existentially quantified variable r in which the ∇_v^C do not occur. In a unification-based calculus the occurs check will let all those proofs fail where an actual information flow takes places from ∇_v^C to v. The purpose of function arguments for f_v is exactly to retain this crucial property in the abstract version of the calculus. We must make sure that a function f_v – abstracting the effect of α on v – gets at least those variables as arguments that are parts of the term representing the final value of v after α.

Theorem 1 (Soundness). *The rules of the DL calculus given in Figs. 2 and 3 are sound: the root of a closed proof tree is a valid sequent.*

$$
\begin{array}{rcll}
\{\, w_1 := t_1, \ldots, w_k := t_k \,\}\; w_i & \to^{\mathrm{dl}} & t_i & \text{if } w_j \neq w_i \text{ for } i < j \leq k \\
\{\, w_1 := t_1, \ldots, w_k := t_k \,\}\; t & \to^{\mathrm{dl}} & t & \text{if } w_1, \ldots, w_k \notin \mathrm{vars}(t) \\
\{\, U \,\}\; f(t_1, \ldots, t_n) & \to^{\mathrm{dl}} & f(\{\, U \,\}\; t_1, \ldots, \{\, U \,\}\; t_n) & \\
\{\, U \,\}\; (t_1 = t_2) & \to^{\mathrm{dl}} & \{\, U \,\}\; t_1 = \{\, U \,\}\; t_2 & \\
\{\, U \,\}\; \neg\phi & \to^{\mathrm{dl}} & \neg\{\, U \,\}\; \phi & \\
\{\, U \,\}\; (\phi_1 * \phi_2) & \to^{\mathrm{dl}} & \{\, U \,\}\; \phi_1 * \{\, U \,\}\; \phi_2 & \text{for } * \in \{\vee, \wedge\}
\end{array}
$$

$$
\{\, U \,\}\; \{\, w_1 := t_1, \ldots, w_k := t_k \,\}\; \phi \;\to^{\mathrm{dl}}\; \{\, U,\; w_1 := \{\, U \,\}\; t_1, \ldots, w_k := \{\, U \,\}\; t_k \,\}\; \phi
$$

Fig. 3. Application rules for updates in dynamic logic, as far as they are required for Lem. 6. Further application and simplification rules are necessary in general.

Simulating Type Derivations in the DL Calculus. In order to show subsumption of the type system in the logic, we first put the connection between invariance sets and context on solid ground. It suffices to approximate the invariance of variables v with the requirement that v must not occur as left-hand side of assignments ($\mathit{Lhs}(\alpha)$ is the set of all left-hand sides of assignments in α).

Lemma 2. *In the type system of [12], see Fig. 1, the following equivalence holds:*

$$p \vdash^{\mathrm{HS}} \nabla \;\{\, \alpha \,\}\; \nabla' \quad \textit{iff} \quad \bot \vdash^{\mathrm{HS}} \nabla \;\{\, \alpha \,\}\; \nabla' \;\; \textit{and} \;\; \textit{f.a. } v \in \mathit{Lhs}(\alpha) : \; p \sqsubseteq \nabla'(v)$$

Furthermore, we can normalize type derivations thanks to the Canonical Derivations Lemma of [12]. The crucial ingredient is the concretisation operator γ^* defined in (3).

Lemma 3 (Canonical Derivations).

$$\bot \vdash^{\mathrm{HS}} \nabla \;\{\, \alpha \,\}\; \nabla' \quad \textit{iff} \quad \bot \vdash^{\mathrm{HS}} \Delta_0 \;\{\, \alpha \,\}\; \gamma^*_{\nabla}(\nabla') \qquad \textit{where } \Delta_0 = \lambda x.\, \{x\}$$

For brevity, we must refer to Hunt and Sands' paper for details, but in the setting at hand one can intuitively take Lemma 3 as stating that any typing judgment can also be understood as a dependency judgment: the typing on the left-hand side is equivalent to the statement that the final value of x *may depend on* the initial value of y only if y appears in the post-type, or dependence set, $\gamma^*_{\nabla}(\nabla')(x)$.

The type system of Fig. 4 only mentions judgments with a pre-type Δ_0 as depicted on the right-hand side of the equivalence in Lemma 3. Further, the context p has been replaced by equivalent side conditions (Lemma 2), and rule $\textsc{Seq}^{\mathrm{HS}}$ is built into the other rules, i.e., the rules always work on the initial statement of a program. Likewise, rule $\textsc{Sub}^{\mathrm{HS}}$ has been integrated in $\textsc{Skip}^{\mathrm{cf}}$ and $\textsc{While}^{\mathrm{cf}}$. The type system is equivalent to Hunt and Sands' system (Fig. 1):

Lemma 4.

$$\bot \vdash^{\mathrm{HS}} \Delta_0 \;\{\, \alpha \,\}\; \nabla \qquad \textit{if and only if} \qquad \vdash^{\mathrm{cf}} \Delta_0 \;\{\, \alpha \,\}\; \nabla$$

The proof proceeds in multiple steps by devising intermediate type systems, each of which adds a modification towards the system in Fig. 4 and which is equivalent to Hunt and Sands' system.

Obviously, due to the approximating character of $\textsc{If}^{\mathrm{dl}}$ and $\textsc{While}^{\mathrm{dl}}$ (and the lack of arithmetic), our DL calculus is not (relatively) complete in the sense of [11]. For the particular judgements $\{\ \alpha\ \} \Downarrow (\nabla, I)$ the calculus is, however, not more incomplete than the type system of Fig. 1: every typable program can also be proven secure using the DL calculus.[6]

Theorem 5.

$$\bot \vdash^{\mathrm{HS}} \Delta_0 \{\ \alpha\ \}\ \nabla \qquad \textit{implies} \qquad \vdash^{\mathrm{dl}}\ \{\ \alpha\ \} \Downarrow (\nabla, \emptyset)$$

The proof of the theorem is constructive: A method for translating type derivations into DL proofs is given. The existence of this translation mapping shows that proving in the DL calculus is in principle not more difficult than typing programs using the system of Fig. 1.

The first part of the translation is accomplished by Lem. 4, which covers structural differences between type derivations and DL proofs. Applications of the rules of Fig. 4 can then almost directly be replaced with the corresponding rules of the DL calculus:

Lemma 6.

$$\vdash^{\mathrm{cf}} \Delta_0 \{\ \alpha\ \}\ \nabla \qquad \textit{implies} \qquad \vdash^{\mathrm{dl}}\ \{\ \alpha\ \} \Downarrow (\nabla, \emptyset)$$

5 Higher Precision and Delimited Information Release

Many realistic languages feature exceptions as a means to indicate failure. The occurrence of an exception can also lead to information leakage. Therefore, an information flow analysis for such a language must, at each point where an exception might possibly occur, either ensure that this will indeed not happen at runtime or verify that the induced information flow is benign. The Jif system [15] which implements a security type system for a large subset of the Java language employs a simple data flow analysis to retain a practically acceptable precision w.r.t. exceptions. The data flow analysis can verify the absence of null pointer exceptions and class cast exceptions in certain cases. However, to enhance the precision of this analysis to an acceptable level one is forced to apply a slightly cumbersome programming style.

The need for treatment of exceptions is an example showing that we actually gain something from the fact that our analysis is embedded in a more general program logic: there is no need to stack one analysis on top of the other to scale

[6] The converse of Theorem 5 does not hold. In the basic version of the calculus of Fig. 2, untypable programs like "**if** (h) $\{l = 1\}$ $\{l = 0\}$" can be proven secure. Sect. 5 discusses how the precision of the DL calculus can be further augmented.

$$\dfrac{}{\vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \}\; \nabla}\ \textsc{Skip}^{\mathrm{cf}} \qquad\qquad v \in \nabla(v) \text{ for all } v \in \mathrm{PVar}$$

$$\dfrac{\mathit{\Delta}_0 \vdash E : t \qquad \vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \dots\ \}\; \gamma^*_{\mathit{\Delta}_0[v \mapsto t]}(\nabla)}{\vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ v = E\ ;\ \dots\ \}\; \nabla}\ \textsc{Assign}^{\mathrm{cf}}$$

$$\dfrac{\begin{array}{c}\mathit{\Delta}_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \dots\ \}\; \gamma^*_{\nabla}(\nabla') \\ \vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \alpha_i\ \}\; \nabla \quad (i = 1,2)\end{array}}{\vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \mathbf{if}\ b\ \alpha_1\ \alpha_2\ ;\ \dots\ \}\; \nabla'}\ \textsc{If}^{\mathrm{cf}} \qquad \begin{array}{l}\text{f.a. } v \in \mathit{Lhs}(\alpha_1).\ t \sqsubseteq \nabla(v) \\ \text{f.a. } v \in \mathit{Lhs}(\alpha_2).\ t \sqsubseteq \nabla(v)\end{array}$$

$$\dfrac{\begin{array}{c}\mathit{\Delta}_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \dots\ \}\; \gamma^*_{\nabla}(\nabla') \\ \vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \alpha\ \}\; \gamma^*_{\nabla}(\nabla)\end{array}}{\vdash^{\mathrm{cf}} \mathit{\Delta}_0 \;\{\ \mathbf{while}\ b\ \alpha\ ;\ \dots\ \}\; \nabla'}\ \textsc{While}^{\mathrm{cf}} \qquad \begin{array}{l}v \in \nabla(v) \text{ for all } v \in \mathrm{PVar} \\ \text{f.a. } v \in \mathit{Lhs}(\alpha).\ t \sqsubseteq \gamma^*_{\nabla}(\nabla)(v)\end{array}$$

Fig. 4. Intermediate flow-sensitive type system for information flow analysis

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c} * \\ [\, f_l'(\mathit{TRUE}) \equiv R \,]\end{array}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ f_l'(\mathit{TRUE}) = R}\ \textsc{close-eq}^{\mathrm{dl}}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ f_l'(\mathit{odd}(f_h(R))) = R}\ \textsc{apply-eq}^{\mathrm{dl}}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ \{\, l := f_l(R), h := f_h(R) \,\}\ \{\, l := f_l'(\mathit{odd}(h)) \,\}\ l = R}\ \xrightarrow{*}{}^{\mathrm{dl}}}{\mathcal{D}}$$

$$\dfrac{\cdots \qquad \dfrac{\dfrac{\dfrac{\dfrac{*}{\vdash^{\mathrm{dl}}\ \{\, l = 0 \,\} \Downarrow (\nabla, \{h\})} \qquad \dfrac{*}{\vdash^{\mathrm{dl}}\ \{\, l = 1 \,\} \Downarrow (\nabla, \{h\})} \qquad \mathcal{D}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ \{\, l := f_l(R), h := f_h(R) \,\}\ [\alpha]\, l = R}\ \textsc{If}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \{\, l := f_l(R), h := f_h(R) \,\}\ (\mathit{odd}(h) \to [\alpha]\, l = R)}\ \xrightarrow{*}{}^{\mathrm{dl}}, \textsc{imp-right}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \exists r.\ \dot\forall l.\ \dot\forall h.\ (\mathit{odd}(h) \to [\alpha]\, l = r)}\ \textsc{ex-right}^{\mathrm{dl}}, \textsc{all-right}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \{\ \alpha\ \} \Downarrow (\nabla, \{h\}, \mathit{odd}(h))}\ (\mathrm{Def}), \textsc{and-right}^{\mathrm{dl}}$$

Fig. 5. Non-interference proof with delimited information release: The precondition $\mathit{odd}(h)$ entails that (only) the parity of h is allowed to leak into l. A similar proof is required for $\neg \mathit{odd}(h)$. For sake of brevity, we use *odd* both as function and predicate, and only in one step ($\textsc{apply-eq}^{\mathrm{dl}}$) make use of the fact that $\mathit{odd}(f_h(R))$ actually represents the equation $\mathit{odd}(f_h(R)) = \mathit{TRUE}$.

the approach up to larger languages, but we can coherently deal with added features, in this case exceptions, within one calculus. In the precise version of the calculus for JavaCard – as implemented in the KeY system [1] – exceptions are handled like conditional statements by branching on the condition under which an exception would occur. An uncaught exception is treated as non-termination. As an example, the division v_1/v_2 would have the condition that v_2 is zero ("$..\ ...$" denotes a context possibly containing exception handlers):

$$\frac{v_2 \neq 0 \ \vdash^{\mathrm{dl}} \ \{\, w := v_1/v_2 \,\}\, [..\ \ ...]\, \phi \qquad v_2 = 0 \ \vdash^{\mathrm{dl}} \ [..\ \textbf{throw}\ E\ ...]\, \phi}{\vdash^{\mathrm{dl}} \ [..\ w = v_1/v_2\ ...]\, \phi} \quad .$$

If we knew $v_2 \neq 0$ at this point of the proof, implying that the division does in fact not raise an exception, the right branch could be closed immediately. Because our DL calculus stores the values of variables (instead of only the type) as long as no abstraction occurs, this information is often available: (i) rule $\textsc{Assign}^{\mathrm{dl}}$ does not involve abstraction, which means that sequential programs can be executed without loss of information, and (ii) invariance sets I in non-interference judgments allow to retain information about unchanged variables also across conditional statements and loops.

This can be seen for a program like "$v = 2\ ;\ \textbf{while}\ b\ \alpha\ ;\ w = w/v$" in which α does not assign to v. By including v in the invariance set for "$\textbf{while}\ b\ \alpha$" we can deduce $v = 2$ also after the loop, and thus be sure that the division will succeed. This is a typical example for a program containing an initialisation part that establishes invariants, and a use part that relies on the invariants. The pattern recurs in many flavours: examples are the initialisation and use of libraries and the well-definedness of references after object creation. We are optimistic to gather empirical evidence of our claim that the increased precision is useful in practice through future experiments.

Increasing Precision. While our DL calculus is able to maintain state information *across* statements, the rules $\textsc{If}^{\mathrm{dl}}$ and $\textsc{While}^{\mathrm{dl}}$ lose this information in the first premisses, containing non-interference proofs for the statement *bodies*. This makes it impossible to deduce that no exceptions can occur in the program "$v = 2\ ;\ \textbf{while}\ b\ \{w = w/v\}$". As another shortcoming, the branch predicate is not taken into account, so that absence of exceptions cannot be shown for a program like "$\textbf{if}\ (v \neq 0)\ \{w = 1/v\}$ ".

One way to remedy these issues might be to relax the first premisses in $\textsc{If}^{\mathrm{dl}}$ and $\textsc{While}^{\mathrm{dl}}$. The idea is to generalise non-interference judgments and introduce *preconditions* ϕ under which the program must satisfy non-interference.

$$\{\ \alpha\ \} \Downarrow (\nabla, I, \phi) \quad \equiv_{\mathrm{def}} \quad \bigwedge_{v \in \mathrm{PVar}} \begin{cases} \dot{\forall} v_1 \cdots v_n.\ (\phi \rightarrow [\alpha]\, v = u) & , v \in I \\ \dot{\forall} \nabla_v.\ \exists r.\ \dot{\forall} \nabla_v^C.\ (\phi \rightarrow [\alpha]\, v = r) & , v \notin I \end{cases}$$

In an extended rule for if-statements, for instance, such a precondition can be used to 'carry through' side formulae and state information contained in the update U, as well as to integrate the branch predicates: we may assume arbitrary preconditions ϕ_1, ϕ_2 in the branches if we can show that they hold before the if-statement:

$$\frac{\begin{array}{cc} \vdash^{\mathrm{dl}} \{\, \alpha_1 \,\} \Downarrow (\nabla, I, \phi_1) & \vdash^{\mathrm{dl}} \{\, \alpha_2 \,\} \Downarrow (\nabla, I, \phi_2) \\ \Gamma, \{\, U \,\}\, b = \mathit{TRUE} \vdash^{\mathrm{dl}} \{\, U \,\}\, \phi_1, \Delta & \Gamma, \{\, U \,\}\, b \neq \mathit{TRUE} \vdash^{\mathrm{dl}} \{\, U \,\}\, \phi_2, \Delta \\ \multicolumn{2}{c}{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\} \{\, \mathrm{ifUpd}(b, \nabla, I) \,\}\, [\ldots]\, \phi, \Delta} \end{array}}{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, [\mathbf{if}\ b\ \alpha_1\ \alpha_2\ ;\ \ldots]\, \phi, \Delta}$$

Probably more interestingly, preconditions allow us to handle delimited information release in the style of [9], i.e. situations in which non-interference does not strictly hold and some well-defined information about secret values may be released. Fig. 5 shows parts of a non-interference proof with delimited information release for the program "$\alpha = \mathbf{if}\ (\mathit{odd}(h))\ \{l = 0\}\ \{l = 1\}$", in which one can learn the parity of h by reading l. The typing ∇ is given by $\nabla(l) = \emptyset, \nabla(h) = \{h\}$, indicating that only declassified information flows into l.

6 Conclusion, Related and Future Work

In this paper we made a formal connection between type-based and logic-based approaches to information flow analysis. We proved that every program that is typeable in Hunt & Sands' type system [12] has a proof in an abstract version of dynamic logic whose construction is not more expensive than the type check. We argued that an integrated logic-based approach fits well into a proof-carrying code framework for establishing security policies of mobile software. In order to support this claim we showed how to increase the precision of the program logic, for example, to express declassification.

Related Work. The background for our work are a number of recent type-based and logic-based approaches to information flow [20, 2, 9, 12]. Our concrete starting points were the flow-sensitive type system of Hunt & Sands [12] and the characterisation of non-interference in [9]. Amtoft & Banerjee [2] devised an analysis with a very logic-like structure, that is however not more precise than the type system by Hunt & Sands. In an early paper Andrews & Reitman [3] developed a flow logic – one may also consider it a security type system – for proving information flow properties of concurrent Pascal programs. They outline a combination of their flow logic with regular Hoare logic, but keep the formulae for both logics separated. Joshi & Leino [13] give logical characterisations of the semantic notion of information flow, and their presentation in terms of Hoare triples is similar in spirit to our basic formulation. Their results do, however, not provide means to aid automated proofs of these triples. Finally, Beringer et al. [7] presented a logic for resource consumption whose proof rules and judgements are derived from a more general program logic; both logics are formalised in the Isabelle/HOL proof assistant. Their approach is similar in spirit to the one presented here, since the preciseness of their derived logic is compared to an extant type system for resource comsumption.

Future Work. On a technical level, we have not investigated the complexity of the translation of HS type derivations to DL proofs (Theorem 5) and the size of resulting proofs in detail. We believe that both can be linear in the size of type derivations, although this requires a more efficient version of proof obligations $\{ \alpha \} \Downarrow (\nabla, I)$. Conceptually, the present work is only a starting point in the integration of type-based and logic-based information-flow analysis. In addition to non-interference and declassification, more complex security policies need to be looked at. It has to be seen how well the notion of abstraction presented in this paper is suited to express these. We also want to extend the program logic to cover at least JavaCard, based on the axiomatisation in [6], as implemented in our program verifier KeY. Ideas towards this goal have been worked out in [18], parts of which are also presented in [10]. Finally, a suitable notion of proof certificate and proof checking for proof-carrying code must be derived for dynamic logic proofs of security policies. This is a substantial task to which a whole Work Package within MOBIUS is devoted.

Acknowledgments

We would like to thank Dave Sands for inspiring discussions and Andrei Sabelfeld for reminding us of declassification. Thanks to Tarmo Uustalu for pointing out [3].

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool: integrating object oriented design and formal verification. *Software and System Modeling*, 4(1):32–54, 2005.

[2] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *11th Static Analysis Symposium (SAS), Verona, Italy*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.

[3] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, Jan. 1980.

[4] A. W. Appel. Foundational Proof-Carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, Los Alamitos, CA, June 2001. IEEE Computer Society.

[5] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114, Pacific Grove,USA, June 2004. IEEE Press.

[6] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.

[7] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004, Montevideo, Uruguay*, volume 3452, pages 347–362. Springer-Verlag, 2005.

[8] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In A. Voronkov, editor, *Proc. 18th International Conference on Automated Deduction CADE, Copenhagen, Denmark*, volume 2392 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[9] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.

[10] R. Hähnle, J. Pan, P. Rümmer, and D. Walter. On the integration of security type systems into program logics. Technical report, Chalmers University of Technology, 2006. Preliminary version at `www.cs.chalmers.se/~philipp/IflowPaper.pdf`.

[11] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, Oct. 2000.

[12] S. Hunt and D. Sands. On flow-sensitive security types. In *Symp. on Principles of Programming Languages (POPL)*. ACM Press, 2006.

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

[14] MOBIUS Project Deliverable D 1.1, Resource and Information Flow Security Requirements, Mar. 2006.
URL: mobius.inria.fr/twiki/pub/DeliverablesList/WebHome/Deliv1-1.pdf

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

[16] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 61–91. Springer-Verlag, 1998.

[17] G. C. Necula and R. R. Schneck. A sound framework for untrustred verification-condition generators. In *Proc. IEEE Symposium on Logic in Computer Science LICS, Ottawa, Canada*, pages 248–260. IEEE Computer Society, 2003.

[18] J. Pan. A theorem proving approach to analysis of secure information flow using data abstraction. Master's thesis, Chalmers University of Technology, 2005.

[19] P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006.

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

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