Formally Verifying Information Flow Type Systems for Concurrent and Thread Systems*

Gilles Barthe
INRIA Sophia-Antipolis, France
Gilles.Barthe@inria.fr

Leonor Prensa Nieto†
LORIA, France
Leonor.Prensa@loria.fr

ABSTRACT

Information flow type systems provide an elegant means to enforce confidentiality of programs. Using the proof assistant Isabelle/HOL, we have machine-checked a recent work of Boudol and Castellani [4], which defines an information flow type system for a concurrent language with scheduling, and shows that typable programs are non-interferent. As a benefit of using a proof assistant, we are able to deal with a more general language than the one studied by Boudol and Castellani. The development constitutes to our best knowledge the first machine-checked account of non-interference for a concurrent language.

Categories and Subject Descriptors

F.3.3 [**Studies of Program Constructs**]: Type structure; D.2.8 [**Software Engineering**]: Software/Program Verification; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages

General Terms

Security, languages, verification

Keywords

Noninterference, concurrency, machine-checked proofs

1. INTRODUCTION

1.1 Background

Security models for mobile and embedded code, such as the Java Virtual Machine and the Common Language Runtime, partially guarantee the innocuity of downloaded applications, but are too weak to enforce strong security. For example, these security models ensure that applications will not perform illegal memory accesses, but fail to guarantee that confidential data shall not be accessed by an unauthorized party. In fact, conventional mechanisms such as static bytecode verification and access control mechanisms are not appropriate to prevent attacks by untrusted code, and platforms for mobile and embedded code should be endowed with appropriate mechanisms that guarantee end-to-end security. One such mechanism for confidentiality is provided by information flow type systems, which track the flow of information in a program execution, and provide a means to enforce statically that no information leakage will result from executing the program. More precisely, information flow type systems guarantee non-interference [8], a high-level property that characterizes programs whose execution does not reveal information about secret data, see [23] for a recent survey.

The definition of non-interference is conditioned by the attacker model which describes the capabilities of the attacker, concerning for instance which observations it can make. In a sequential setting, there are several well-established variants of non-interference. These variants all assume a separation between secret inputs and public inputs on the one hand, and secret outputs and public outputs on the other hand. On the basis of this separation, definitions of non-interference require that the value of public outputs does not depend on the value of secret inputs (termination-insensitive non-interference), or that the termination of the program and the value of public outputs does not depend on the value of secret inputs (termination-sensitive non-interference), or that the execution time of the program and the value of public outputs does not depend on the value of secret inputs (timing-sensitive non-interference).

While information flow type systems provide an effective means to enforce end-to-end confidentiality for sequential programs, concurrency raises a number of subtle issues, starting from the definition of non-interference in a concurrent setting.

A first difficulty is to handle non-determinism. One obvious possibility is to adopt a so-called possibilistic notion of non-interference [4, 27] which considers the set of possible outputs of a program execution (instead of the output in a sequential setting). However, programs that are deemed secure by possibilistic non-interference are subject to refinement attacks: indeed, using a scheduler to execute non-deterministic programs may result in secure programs leaking confidential information. For example, the program

$$(\text{if } h = 0 \text{ then skip else sleep}(100)); l := 0 || l := 1$$

*Work partially supported by the IST Project Profundis and by the ACI Sécurité SPOPS.

†Most of this work was performed while at INRIA Sophia-Antipolis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FMSE'04, October 29, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-961-6/04/0010 ...$5.00.

is likely to terminate with $l = 0$ if $h = 0$ and a round robin scheduler is used.

In order to avoid such refinement attacks, several approaches have been developed to account for schedulers. One approach is to focus on probabilistic non-interference, which deals with probabilistic parallel composition (or a generalization of it that allows to compose an arbitrary number of programs in parallel) and considers the probability of distribution for the possible outputs of a program execution. For example, one can show probabilistic non-interference of programs assuming that probabilities in parallel composition are uniform [26, 30]. Another alternative is to adopt a stronger, scheduler independent, notion of security; for example, one can isolate a large class of schedulers, potentially probabilistic, and require programs to be secure for all the schedulers in this class [25]. Yet another possibility is to extend the programming language with primitives for scheduling, as e.g. in [4, 15]. In this approach, scheduling policies are represented by a concurrent program that is type-checked using the same rules as other concurrent programs. (Such a scenario of schedule-carrying code has been pursued independently in the context of embedded systems [10].)

One further issue with concurrency is the attacker model. While definitions of non-interference in sequential settings can be concerned with an input/output view of the program behavior, definitions of non-interference in concurrent setting usually adopt a more conservative approach in order to prevent that a malicious thread can observe the behavior of other threads and adapts its behavior accordingly. Consequently, definitions of non-interference for concurrent programs often aim at guaranteeing that the confidential data is protected throughout the entire program execution. To this end, such definitions are based on different notions of bisimulation. In particular, bisimulation based notions of non-interference have been adopted in many works on non-interference that have been pursued in the context of process algebra, see e.g. [7], π-calculus, see e.g. [9, 11], and ambient calculus, see e.g. [6]. However, different definitions of non-interference are also considered in the literature: for example, Zdancewic and Myers [32] have recently considered a notion of observational determinism that uses execution traces and considers a complex notion of indistinguishability involving equivalence of (projections along variables of) traces up to prefixing and stuttering.

1.2 Our work

Non-interference for sequential and concurrent languages is very intricate, both in the definition of the type system used for the analysis, and in the soundness proof which establishes that typable programs are non-interferent. It is therefore natural to resort to proof assistants for managing the complexity of the definitions and proofs involved in establishing non-interference for languages.

The purpose of this paper is to report on an experiment with the mechanical verification of a type system for a concurrent language with scheduling, using the Isabelle proof assistant [18]. The programming language is an extension of a simple WHILE language with:

- a parallel composition operator with an interleaving semantics;
- primitive operators for scheduling and synchronization.

Our language is inspired from [4], but features unrestricted sequential composition, whereas [4] requires that the first process of a sequential composition is sequential. We also modify and extend the semantics rules to give a reasonable meaning to all possible programs in our language.

The type system is also taken from [4], and keeps track of the level of loop guards, as well as of the level of assignments. However, we show its validity for our more general language.

Validity, which is expressed as in possibilistic terms and cast in terms of bisimulation, is established against the same notion of security than in [4]: concretely, we define a program P to be non-interferent if $P \approx P$, where $\approx$ is the largest bisimulation on programs. (Note that the notion of bisimulation on programs is drawn from an appropriate notion of bisimulation on configurations, and that, while bisimulation on configurations is an equivalence relation, bisimulation on programs is not.)

Furthermore, our proofs mostly use definitions and proof techniques from [4]; in particular for our main results, we rely on exhibiting an appropriate bisimulation relation $\mathcal{R}$ such that $P \; \mathcal{R} \; P$ for every typable program P. However, as a result of allowing for unrestricted sequential composition, we do not need to prove sequential non-interference prior to proving concurrent non-interference.

Our work not only indicates that proof assistants are mature for verifying state-of-the-art type systems for information flow, but also shows that proof assistants help reduce the complexity of proofs and thereby allow to discard convenient, but unnecessary assumptions in proofs. Perhaps less importantly, our work reveals minor flaws in the definitions and proofs of [4].

1.3 Related work on formal proofs of non-interference

There is a large body of work in machine-checked programming language semantics, in particular in the area of type systems. However, most works in this area focus on "traditional" type systems. In contrast, few formalizations focus on language-based non-interference and in particular, we do not know of any machine-checked proof of non-interference for a concurrent programming language.

Existing works include a formalization of unwinding theorems for intransitive non-interference by Rushby [22], as well as recent formalizations of information flow type systems for a fragment of Java [2, 1]. The latter formalizations have been conducted by Naumann [17] and by Strecker [28], in PVS and Isabelle respectively.

1.4 Contents of the paper

The remainder of the paper is organized as follows. Section 2 gives an introduction to Isabelle/HOL. Section 3.1 presents the concurrent programming language, and its associated type system. Section 3.2 establishes non-interference of the concurrent language. In Section 4.1, we extend the language with scheduling primitives, and endow it with appropriate typing rules. Section 4.2 is devoted to a proof of non-interference for the extended language. We conclude in Section 5 with directions for further work.

2. ISABELLE/HOL

Isabelle [18] is a generic interactive theorem prover which

can be instantiated with different object logics. Isabelle/HOL is the instance for Higher-Order Logic.

We use two base types (*bool* and *nat*) and construct others by type constructors like *list*, *set* or the product type ($\times$) and by function application ($\Rightarrow$). Isabelle also supports inductive definitions of data types (keyword **datatype**).

List notation is similar to ML. The ith component of a list xs is written $xs!i$. The functional $map :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta\ list$ applies a function to all elements of a list. The syntax $xs[i := x]$ denotes the list xs with the ith component replaced by x.

Isabelle distinguishes between object level ($\longrightarrow$) and meta-level ($\Longrightarrow$) implication, and similarly for universal quantification, but this distinction is unimportant for our purposes. The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ represents an implication with assumptions $A_1, \ldots, A_n$ and conclusion A.

All functions must be explicitly declared (keyword **consts**). Non-recursive definitions are defined via the meta-equality ($\equiv$), recursive ones are introduced by **primrec**. The rules for inductively defined sets are introduced by **inductive**. For each inductive definition Isabelle generates the corresponding induction principle (*rule induction*) and the case analysis principle (*rule inversion*).

The statements we prove are preceded by **lemma** or **theorem**, with no formal difference between them. Isabelle provides powerful proof tactics based on rewriting and the classical reasoner, which implements a tableau based prover for predicate logic and sets.

Isabelle has been extensively used for the formalization of programming language semantics, and in particular for the formalization of the Java language, see e.g. [12, 19].

3. NON-INTERFERENCE FOR CONCURRENT SYSTEMS

3.1 The Language and Type System

We start by defining the memory as a mapping from locations to values. The type of locations and values is left unspecified (we can introduce new types in Isabelle by a *type declaration*, which merely introduces its name).

typedecl *loc*
typedecl *val*
types *memory* = *loc* $\Rightarrow$ *val*

Another possibility, which allows for different implementations of memories without affecting much the formalization, is to introduce memories as an abstract type, equipped with lookup and update functions that are assumed to satisfy the expected equational properties. Such an approach has the advantage of avoiding the use of higher-order functions (the function @textaexp below is higher-order), but it is of no concern to us here, since Isabelle supports such functions.

Processes are built up from arithmetic and boolean expressions. Following [4, Assumption 3.2], we assume that expressions always evaluate to a result. Since Isabelle is a logic of total functions, it can be done simply by treating expressions as functions from memories to results.

types *aexp* = *memory* $\Rightarrow$ *val*
types *bexp* = *memory* $\Rightarrow$ *bool*

Another possibility, which is slightly more abstract, is to introduce arithmetic expressions and boolean expressions as abstract types, and define evaluation functions that take as argument a memory and an arithmetic expression (resp. a boolean expression) and return as result a value (resp. a boolean).

The program syntax is defined via a **datatype** definition. Our language is more general than the one presented in [4], where programs of the form $(P \| Q);;\ R$ are not allowed.

datatype *par* =

Skip	(**skip**)
| *Assign loc aexp*	(- :== -)
| *Seq par par*	(-;; -)
| *Cond bexp par par*	(**if** - **then** - **else** -)
| *While bexp par*	(**while** - **do** -)
| *Par par par*	(- ‖ -)

Enclosed in parentheses we give concrete syntax for each construct. We use :== for assignments and ;; for sequential composition to avoid clashes with the predefined := and ; of Isabelle.

The semantics of commands, shown in figure 1, is inductively defined via transition rules between configurations. A configuration is a pair (P, μ) where P is a program and μ is the memory. We use a readable infix syntax for transitions rules (most of the definitions presented in this paper are endowed with a readable infix syntax whose formal declaration is not always explicitly shown here). In the rule for *Assign*, the expression $\mu[l \mapsto v]$ stands for memory update and is defined as

$$\lambda x.\ if\ x{=}l\ then\ v\ else\ \mu\ x$$

The rule *Seq*1 replaces the rule

$$(P, \mu) \longrightarrow_1 (P', \mu') \Longrightarrow (\textbf{skip};;\ P, \mu) \longrightarrow_1 (P', \mu')$$

used in [4]. This modification has two advantages: it solves a minor flaw in the proof of noninterference presented in [4] and more importantly, enables reduction of programs of the form **skip**;; **skip**;; ... ;; **skip**, which would otherwise be irreducible. We also add rules *ParL*1 and *ParR*1 giving thus a reasonable semantics to all programs of type *par*. In particular, programs of the form $(P \| Q);;\ R$ can be reducible in our system.

The types of data and expressions are *security levels*. In [4] they are modeled as elements of a lattice. For our purposes, it suffices to declare a new type *level* as an instance of the axiomatic class *partial-order*, which is a predefined type class with three axioms: reflexivity, transitivity and antisymmetry (w.r.t. a binary relation "$\sqsubseteq$"). Note that we eliminate the need to have meet and join of security levels by modifying the typing rules in an appropriate fashion.

Type judgments are of the form $\vdash P \triangleright t\ s$, where t is a *lower bound* on the level of the assigned variables of P and s is the *guard type*, i.e. an *upper bound* on the level of the loop and conditional guards occurring in P. In [4] the context Γ is a mapping from variables to security levels. Our formalization leaves contexts unspecified and assumes instead the existence of a function that extracts the security level of locations.

consts *getlevel* :: *loc* $\Rightarrow$ *level*

The type system is relative to functions *seca* and *secb* that provide the security level of arithmetic and boolean expressions, respectively.

consts *seca* :: *aexp* ⇒ *level*
consts *secb* :: *bexp* ⇒ *level*

It is inductively defined by the set of inference rules shown in figure 2. To simplify proofs, we formalize syntax-directed rules which already include the necessary subtyping relations. This avoids the use of the meet and join operators as well as dealing with the subtyping rule proper, which we can easily derive from the system above.

lemma *Subtyping*:
$[\![\vdash P \triangleright [t, s];\ t' \sqsubseteq t;\ s \sqsubseteq s']\!] \Longrightarrow \vdash P \triangleright [t', s']$

The definition of non-interference presupposes a set of low security levels which is *downward-closed*.

consts L :: *level* ⇒ *bool*
axioms *Ldown*: $[\![L\ x';\ x \sqsubseteq x']\!] \Longrightarrow L\ x$

Equality on memories is defined relative to L.

constdefs
eqmem :: *memory* ⇒ *memory* ⇒ *bool* ((- ≃ -))
$\mu \simeq \mu' \equiv \forall x.\ L\ (getlevel\ x) \longrightarrow \mu\ x = \mu'\ x$

In the sequel we assume that the security level of expressions is correct in the sense that evaluating a low expression with low equal memories should yield the same result. Such an assumption corresponds to [4, Assumption 3.3.].

axioms
beh-aexp: $[\![\mu \simeq \mu';\ L\ (seca\ a)]\!] \Longrightarrow (a\ \mu) = (a\ \mu')$
beh-bexp: $[\![\mu \simeq \mu';\ L\ (secb\ b)]\!] \Longrightarrow (b\ \mu) = (b\ \mu')$

3.2 Properties of Typed Programs

In this section we formally define and prove some properties of typable programs. A program is *typable* if there exist types such that a typing judgment can be derived in the system.

constdefs *typable* :: *par* ⇒ *bool*
typable $P \equiv \exists t\ s.\ \vdash P \triangleright [t, s]$

The final goal is to establish that typable programs are secure in the sense of non-interference. Some preliminary lemmas are needed. The first one states that types are preserved along execution.

lemma *subject-reduction*:
$[\![\vdash P \triangleright [t, s]\ ;\ (P, \mu) \longrightarrow_1 (P', \mu')]\!] \Longrightarrow \vdash P' \triangleright [t, s]$

Following [4], we use various notions of bisimulation to state and reason about non-interference. First we define the relation $\twoheadrightarrow$ as the reflexive closure of $\longrightarrow_1$. It is defined as a set of pairs of configurations inductively generated by two rules:

execr-refl: $cf \twoheadrightarrow cf$
execr-inj: $cf \longrightarrow_1 cf' \Longrightarrow cf \twoheadrightarrow cf'$

We say that Q is a *derivative* of P, written $P \rightsquigarrow Q$, if we can deduce it from the following rules:

der-refl: $P \rightsquigarrow P$
der-step: $[\![(P, \mu) \longrightarrow_1 (P', \mu');\ P' \rightsquigarrow Q]\!] \Longrightarrow P \rightsquigarrow Q$

(Semantically) high programs are programs that never modify the low part of the memory. We formalize this definition and prove some useful properties:

constdefs *ship* :: *par* ⇒ *bool*
ship $P \equiv \forall P'\ \mu\ Q\ \mu'.$
$(P \rightsquigarrow P' \wedge (P', \mu) \longrightarrow_1 (Q, \mu')) \longrightarrow \mu \simeq \mu'$

lemma *ship-sr*: $[\![(P, \mu) \longrightarrow_1 (Q, \mu');\ ship\ P]\!] \Longrightarrow ship\ Q$
lemma *ship-skip*: *ship* **skip**
lemma *ship-seq*: $[\![ship\ P;\ ship\ Q]\!] \Longrightarrow ship\ (P;;\ Q)$
lemma *ship-par*: $[\![ship\ P;\ ship\ Q]\!] \Longrightarrow ship\ (P \parallel Q)$

Observe that with our definition of derivative any program Q that can be obtained by reducing P via the operational semantics, allowing *arbitrary* changes in the memory throughout the reduction, is a derivative of P. This gives us the right definition of high programs. In [4], Q is defined as a *derivative* of P, if for some μ and μ' we have $(P, \mu) \longrightarrow^* (Q, \mu')$, where $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow_1$. With this definition, however, the lemma *ship-sr* is not true. Matos et al. also correct this problem by giving an equivalent definition [15].

The predicate *bis*, shown in figure 3, defines when a relation R on configurations is a *bisimulation*. In this definition, *sym* R means that the relation R is symmetric. (Note that two stronger notions of bisimulation, namely *quasi-strong* and *strong* bisimulations, are defined in [4]. The first one is needed to prove non-interference for the sequential sublanguage. Thanks to our generalization of the language we do not need this definition to prove non-interference of parallel programs. However, both of them will be necessary in section 4.2 when we prove non-interference for scheduled thread systems.)

The domain of bisimulations characterizes *secure* programs.

constdefs
secure :: *par* ⇒ *bool*
secure $P \equiv \exists S.\ (P, P) \in S$
$\wedge$ *bis* $\{((P, \mu), (Q, \nu)).\ (P, Q) \in S \wedge \mu \simeq \nu\}$

The first result establishes bisimilarity of high programs using the relation S_0.

constdefs $S0$ :: (*par* × *par*) *set*
$S0 \equiv \{(P, Q).\ ship\ P \wedge ship\ Q\}$

We define the corresponding relation R_0 between configurations and prove that it is a bisimulation.

constdefs $R0$ :: ((*par* × *memory*) × (*par* × *memory*)) *set*
$R0 \equiv \{((P, s), (Q, t)).\ P\ S_0\ Q \wedge s \simeq t\}$

lemma *R0-is-bis*: *bis R0*

We define *bounded* and *guarded* programs.

constdefs *bounded* :: *par* ⇒ *bool*
bounded $P \equiv \forall t\ s.\ \vdash P \triangleright [t, s] \longrightarrow L\ t$

constdefs *guarded* :: *par* ⇒ *bool*
guarded $P \equiv \exists t\ s.\ \vdash P \triangleright [t, s] \wedge (L\ s)$

Observe that from the typing rule *Seq* and the *Ldown* axiom, we can prove the following property of bounded programs:

lemma *bounded-seq*: *bounded* $Q \Longrightarrow$ *bounded* $(P;;\ Q)$

A program which is not bounded cannot write on variables of low level, and therefore such a program is high.

lemma *notbounded-ship*: ¬ *bounded* $P \Longrightarrow$ *ship* P

$$
\begin{array}{ll}
\text{Assign:} & (x :== a,\ \mu) \longrightarrow_1 (\textsf{skip},\ \mu[x \mapsto (a\ \mu)]) \\
 & \\
\text{Seq1:} & (\textsf{skip};;\ P,\ \mu) \longrightarrow_1 (P,\ \mu) \\
\text{Seq2:} & (P,\ \mu) \longrightarrow_1 (P',\ \mu') \Longrightarrow (P;;\ Q,\ \mu) \longrightarrow_1 (P';;\ Q,\ \mu') \\
 & \\
\text{CondT:} & b\ \mu \Longrightarrow (\textsf{if}\ b\ \textsf{then}\ P\ \textsf{else}\ Q,\ \mu) \longrightarrow_1 (P,\ \mu) \\
\text{CondF:} & \neg b\ \mu \Longrightarrow (\textsf{if}\ b\ \textsf{then}\ P\ \textsf{else}\ Q,\ \mu) \longrightarrow_1 (Q,\ \mu) \\
 & \\
\text{WhileT:} & b\ \mu \Longrightarrow (\textsf{while}\ b\ \textsf{do}\ P,\ \mu) \longrightarrow_1 (P;;\ \textsf{while}\ b\ \textsf{do}\ P,\ \mu) \\
\text{WhileF:} & \neg b\ \mu \Longrightarrow (\textsf{while}\ b\ \textsf{do}\ P,\ \mu) \longrightarrow_1 (\textsf{skip},\ \mu) \\
 & \\
\text{ParL1:} & (\textsf{skip} \parallel Q,\ \mu) \longrightarrow_1 (Q,\ \mu) \\
\text{ParL2:} & (P,\ \mu) \longrightarrow_1 (P',\ \mu') \Longrightarrow (P \parallel Q,\ \mu) \longrightarrow_1 (P' \parallel Q,\ \mu') \\
 & \\
\text{ParR1:} & (P \parallel \textsf{skip},\ \mu) \longrightarrow_1 (P,\ \mu) \\
\text{ParR2:} & (Q,\ \mu) \longrightarrow_1 (Q',\ \mu') \Longrightarrow (P \parallel Q,\ \mu) \longrightarrow_1 (P \parallel Q',\ \mu')
\end{array}
$$

Figure 1: Operational semantics.

$$
\begin{array}{ll}
\text{Skip:} & \vdash \textsf{skip} \triangleright [t,\ s] \\
 & \\
\text{Assign:} & [\![\ \text{seca}\ a \sqsubseteq \text{getlevel}\ x;\ t \sqsubseteq \text{getlevel}\ x\ ]\!] \Longrightarrow \vdash x{:=}{=}a \triangleright [t,\ s] \\
 & \\
\text{Seq:} & [\![\ \vdash P \triangleright [t,\ s];\ \vdash Q \triangleright [t',\ s'];\ s \sqsubseteq t';\ t'' \sqsubseteq t;\ t'' \sqsubseteq t';\ s \sqsubseteq s'';\ s' \sqsubseteq s''\ ]\!] \\
 & \Longrightarrow \vdash P;;\ Q \triangleright [t'',\ s''] \\
 & \\
\text{Cond:} & [\![\ \vdash P \triangleright [t,\ s];\ \vdash Q \triangleright [t,\ s];\ \text{secb}\ b \sqsubseteq t;\ \text{secb}\ b \sqsubseteq s';\ s \sqsubseteq s';\ t' \sqsubseteq t\ ]\!] \\
 & \Longrightarrow \vdash \textsf{if}\ b\ \textsf{then}\ P\ \textsf{else}\ Q \triangleright [t',\ s'] \\
 & \\
\text{While:} & [\![\ \vdash P \triangleright [t,\ s];\ \text{secb}\ b \sqsubseteq t;\ s \sqsubseteq t;\ \text{secb}\ b \sqsubseteq s';\ s \sqsubseteq s';\ t' \sqsubseteq t\ ]\!] \\
 & \Longrightarrow \vdash \textsf{while}\ b\ \textsf{do}\ P \triangleright [t',\ s'] \\
 & \\
\text{Par:} & [\![\ \vdash P \triangleright [t,\ s];\ \vdash Q \triangleright [t,\ s]\ ]\!] \Longrightarrow \vdash P \parallel Q \triangleright [t,\ s]
\end{array}
$$

Figure 2: Type system for concurrent programs.

$$
\begin{array}{ll}
\textbf{constdefs} & \text{bis} :: (((\text{par} \times \text{memory}) \times (\text{par} \times \text{memory}))\ \text{set}) \Rightarrow \text{bool} \\
\text{bis}\ R \equiv & \text{sym}\ R \\
 & \wedge\ (\forall P\ \mu\ Q\ \nu.\ ((P,\ \mu),\ (Q,\ \nu)) \in R \longrightarrow \mu \simeq \nu \\
 & \wedge\ (\forall P'\ \mu'.\ (P,\ \mu) \longrightarrow_1 (P',\ \mu') \longrightarrow (\exists Q'\ \nu'.\ (Q,\ \nu) \twoheadrightarrow (Q',\ \nu') \wedge ((P',\ \mu'),\ (Q',\ \nu')) \in R)))
\end{array}
$$

Figure 3: Bisimulation for concurrent programs.

However, a high program is not necessarily not bounded (see the counterexample in [4]). The next lemma characterizes the behaviour of guarded programs.

lemma *behaviour-of-guarded-programs*:
$[\![guarded\ P;\ \mu \simeq \nu;\ (P,\ \mu) \longrightarrow_1 (P',\ \mu')]\!]$
$\Longrightarrow \exists \nu'.\ (P,\ \nu) \longrightarrow_1 (P',\ \nu') \wedge \mu' \simeq \nu'$

We now define a relation S_2 on parallel programs which will be the key to the proof of noninterference for concurrent programs. It is inductively defined by the rules of figure 4. We added *clause*4*s*, *clause*5, *clause*5*s*, *clause*6 and *clause*6*s* to the original definition of [4] in order to do the proof for our enriched semantics.

The following three properties are used in the proof of non-interference:

lemma *S2-sym*: $P\ S_2\ Q \Longrightarrow Q\ S_2\ P$
lemma *S2-refl*: $typable\ P \Longrightarrow P\ S_2\ P$
lemma *S2-skip-ship*: **skip** $S_2\ P \Longrightarrow ship\ P$

We now prove the non-interference result for programs of type *par*.

theorem *Concurrent-Noninterference*: $typable\ P \Longrightarrow secure\ P$

By instantiating with the relation S_2 we obtain the following subgoals:

1. $typable\ P \Longrightarrow P\ S_2\ P$
2. $typable\ P \Longrightarrow bis\ \{((P,\ \mu),\ Q,\ \nu).\ P\ S_2\ Q \wedge \mu \simeq \nu\}$

The first subgoal is solved using lemma *S*2-*refl*. It remains to prove that the relation R_2 (the extended relation of S_2 for configurations) is a bisimulation.

lemma *R2-is-bis*: $bis\ R2$

This follows from the following auxiliary lemma on which we apply induction on the derivation of S_2:

lemma *R2-is-bis-aux*: $P\ S_2\ P' \Longrightarrow$
$\forall \mu\ \mu'\ Q\ \nu.\ \mu \simeq \mu' \longrightarrow (P,\ \mu) \longrightarrow_1 (Q,\ \nu)$
$\longrightarrow (\exists Q'\ \nu'.\ (P',\ \mu') \twoheadrightarrow (Q',\ \nu') \wedge (Q,\ \nu)\ R_2\ (Q',\ \nu'))$

The proof for the case concerning the rule *clause*2 had a minor flaw in [4]. This could be solved by adding two new clauses to the definition of S_2: $P\ S_2\ Q \Longrightarrow$ (**skip**;; P) $S_2\ Q$ (and $P\ S_2\ Q \Longrightarrow P\ S_2$ (**skip**;; Q) to preserve symmetry). However, by modifying the rule *Seq*1 of the semantics as explained above this problem is solved more elegantly.

4. NON-INTERFERENCE FOR SCHEDULING SEQUENTIAL PROGRAMS

4.1 The Language and Type System

We consider parallel execution of sequential threads controlled by a scheduler: $Sched\ [\![T_1, \ldots, T_n]\!]$. The scheduler is a parallel program of type *par* and each thread is a program of the following type:

datatype *thread* = *When bexp seq* (**when** - **do** -)

The type *seq* of sequential programs is defined like parallel programs in the previous section without the parallel construct. A thread T is thus a sequential program S with a guard b and concrete syntax **when** b **do** S. The execution of S is allowed to proceed, for one step, when the condition b holds, i.e. execution can be triggered and suspended by the scheduler. A controlled thread system has the following type:

datatype *control* = *Control par* (*thread list*) (-[![-]!])

A controlled thread system $P[\![T]\!]$ is legal iff the variables written in P are disjoint from the variables written in T.

In [4], parallelism of threads is defined via a binary operator ($\|$), which allows us to express parallelism of a concrete number of threads. Using lists we can also reason about parameterized parallel composition of threads. This is not relevant for the formalization of [4] but it is useful for reasoning about concrete programs. Given a thread $T(i)$ depending on a parameter i that varies between 0 and n we can formally express the list $[T(0), \ldots, T(n)]$ using the HOL function *map* and the construct $[0..n]$, which represents the list of natural numbers from 0 to n, i.e.

$$map\ (\lambda i.\ T(i))\ [0..n]$$

Consequently, our formalization proves non-interference also for parameterized systems of threads.

The semantics of *When*-instructions is described by the two rules of figure 5, where $\longrightarrow_s$ is the transition relation for sequential programs. The first rule allows the sequential program S to proceed from μ, for one step, when the condition b holds in μ. The rule *When-op*2 is technically convenient but harmless. It simply allows to ignore a terminated thread.

To define the operational semantics rules for *Control* we need functions that calculate the set of variables that are written by a thread or by the scheduler. The function *wpar* calculates this set for parallel programs. We merely show two clauses:

wpar (**skip**) = {}
wpar (x:== a) = {x}

The definition for the rest of the constructs is obvious. The function *wseq* for sequential programs is analogous to *wpar* for the sequential subset. Finally, writable variables of a thread are those of the sequential body:

wthread (**when** b **do** S) = *wseq* S

The semantics of *Control*-instructions is defined via the two rules shown in figure 4. The execution of *Control-op*1 returns the memory μ with the conjunction of the updates operated by P and T. This is expressed using the Isabelle function *overwrite* defined as

$$f(g|A) \equiv \lambda a.\ if\ a \in A\ then\ g\ a\ else\ f\ a$$

We could also write it in the opposite order, i.e.

$$\mu'\ (\mu'' \mid (wthread\ T!i))$$

Both are equivalent under the restriction that the writable variables of P and T be disjoint.

The typing rules for the new operators are shown in figure 6. They also include the subtyping relations in the premises. The rule for *When* has the premise

$$\vdash_s\ S \triangleright [t,\ s]$$

that corresponds to the typing judgement for the sequential program.

clause1: $[\![\ \text{ship P; ship Q; typable P; typable Q}\ ]\!] \Longrightarrow \text{P } S_2 \text{ Q}$
clause2: $[\![\ \text{bounded P; typable P}\ ]\!] \Longrightarrow \text{P } S_2 \text{ P}$
clause3: $[\![\ \text{P } S_2 \text{ Q; } \neg \text{ bounded R; typable (P;; R); typable (Q;; R)}\ ]\!] \Longrightarrow \text{(P;; R) } S_2 \text{ (Q;; R)}$

clause4: $[\![\ \text{P1 } S_2 \text{ P2; Q1 } S_2 \text{ Q2; typable (P1 } \| \text{ Q1); typable (P2 } \| \text{ Q2)}\ ]\!] \Longrightarrow \text{(P1 } \| \text{ Q1) } S_2 \text{ (P2 } \| \text{ Q2)}$
clause4s: $[\![\ \text{P1 } S_2 \text{ Q2; Q1 } S_2 \text{ P2; typable (P1 } \| \text{ Q1); typable (P2 } \| \text{ Q2)}]\!] \Longrightarrow \text{(P1 } \| \text{ Q1) } S_2 \text{ (P2 } \| \text{ Q2)}$

clause5: $[\![\ \text{ship P; Q1 } S_2 \text{ Q2; typable Q1; typable (P } \| \text{ Q2)}\ ]\!] \Longrightarrow \text{Q1 } S_2 \text{ (P } \| \text{ Q2)}$
clause5s: $[\![\ \text{ship P; Q1 } S_2 \text{ Q2; typable Q1; typable (P } \| \text{ Q2)}\ ]\!] \Longrightarrow \text{Q1 } S_2 \text{ (Q2 } \| \text{ P)}$

clause6: $[\![\ \text{ship P; Q1 } S_2 \text{ Q2; typable Q2; typable (P } \| \text{ Q1)}\ ]\!] \Longrightarrow \text{(P } \| \text{ Q1) } S_2 \text{ Q2}$
clause6s: $[\![\ \text{ship P; Q1 } S_2 \text{ Q2; typable Q2; typable (P } \| \text{ Q1)}\ ]\!] \Longrightarrow \text{(Q1 } \| \text{ P) } S_2 \text{ Q2}$

Figure 4: Relation S_2.

When-op1: $[\![\ \text{b } \mu\text{; (S, } \mu) \longrightarrow_s \text{(S}', \mu')\ ]\!] \Longrightarrow (\textbf{when b do S}, \mu) \longrightarrow_t (\textbf{when b do S}', \mu')$

When-op2: $[\![\ \text{b } \mu\text{; } \neg\ (\exists\, \text{s. (S, } \mu) \longrightarrow_s \text{s)}\ ]\!] \Longrightarrow (\textbf{when b do S}, \mu) \longrightarrow_t (\textbf{when b do S}, \mu)$

Control-op1: $[\![\ \text{(P, } \mu) \longrightarrow_1 \text{(P}', \mu')\text{; i<length T; (T!i, } \mu) \longrightarrow_t \text{(t}', \mu'')\ ]\!]$
$\Longrightarrow \text{(P}[\![\text{T}]\!], \mu) \longrightarrow_c \text{(P}'[\![\text{T[i:=t}']]\!], \mu''\ (\mu' \mid \text{(wpar P)}))$

Control-op2: $[\![\ \text{(P, } \mu) \longrightarrow_1 \text{(P}', \mu')\text{; } \forall \text{i<length T. } \neg(\exists\, \text{s. (T!i, } \mu) \longrightarrow_t \text{s)}\ ]\!]$
$\Longrightarrow \text{(P}[\![\text{T}]\!], \mu) \longrightarrow_c \text{(P}'[\![\text{T}]\!], \mu')$

Figure 5: Operational semantics of the *When* and *Control* instructions.

When: $[\![\ \vdash_s \text{S} \triangleright \text{[t, s]; secb b} \sqsubseteq \text{s}'\text{; secb b} \sqsubseteq \text{t; t}' \sqsubseteq \text{t}\ ]\!] \Longrightarrow \vdash_t (\textbf{when b do S}) \triangleright [\text{t}', \text{s}']$

Control: $[\![\ \vdash \text{P} \triangleright \text{[t, s]; } \forall \text{i<length T. } \vdash_t \text{T!i} \triangleright \text{[t, s]; s} \sqsubseteq \text{t; s} \sqsubseteq \text{s}'\text{; t}' \sqsubseteq \text{t}]\!] \Longrightarrow \vdash_c \text{P}[\![\text{T}]\!] \triangleright [\text{t}', \text{s}']$

Figure 6: Typing rules for the *When* and *Control* instructions.

We conclude this section by briefly remarking that the language features a combination of constructs that is powerful enough to encode a large variety of schedulers. For instance, we can define a round robin scheduler with time slice t for a system with n threads running the programs as follows, or if a suitable random function is aggregated to the language, a uniform scheduler, see [4]. Such examples demonstrate that the approach followed in this paper has the ability to capture realistic scenarios, but we do not claim that it is the sole appropriate approach for securing concurrent programs.

4.2 Properties of Typed Programs

All definitions introduced in section 3.2 are also defined for the new types *seq*, *thread* and *control* with their names preceded by s, t and c, respectively. For instance, *ship* is named *sship* for sequential programs, *tship* for threads and *cship* for controlled systems.

The proof of non-interference for controlled thread systems requires two stronger notions of bisimulation: *strong* and *quasi-strong bisimulation*. The latter is defined both on sequential programs and controlled thread systems.

A relation R between configurations of sequential programs is a *quasi−strong bisimulation* if the predicate *qsbis* R, shown in figure 7, holds. Likewise, a relation

In order to prove our final result, one needs to exhibit a quasi-strong bisimulation R such that $P\ R\ P$ for all typable sequential programs P. To this end, we define a relation S_1 between sequential programs is inductively defined by three rules analogous to *clause*1, *clause*2 and *clause*3 of the relation S_2 in the previous section. We then prove that the corresponding R_1 is a quasi-strong bisimulation.

lemma *R1-is-qsbis*: *qsbis R*1

A relation R is a *strong bisimulation* if it satisfies the conditions of a bisimulation where $\twoheadrightarrow$ is replaced by $\longrightarrow$. The formal definition for the case of programs of type *control*, called *csbis* is shown in figure 8.

The predicate *cqsbis* defines quasi-strong bisimulations for controlled thread systems. It is easy to prove that a strong bisimulation is also a quasi-strong bisimulation.

lemma *csbis-is-cqsbis*: *csbis* $R \Longrightarrow$ *cqsbis* R

In order to prove our final result, we shall exhibit a strong bisimulation R such that $P\ R\ P$ for all typable controlled thread systems P.

We proceed as follows: first, we exhibit a strong bisimulation S_3 on controlled thread systems. Since this relation does not have the expected property, one defines a quasi-strong bisimulation S_4 that extends S_3 and enjoys the property that $P\ S_4\ P$ for all typable controlled thread systems P. Finally, we show that S_4 is a strong bisimulation, by providing a sufficient condition for a quasi-strong bisimulation to be a strong bisimulation, and by showing that S_4 enjoys this property.

Figure 9 shows the definition of a relation S_3 on controlled thread systems. For clarity we use two functions, *guard-of-when* and *seq-of-when*, which given a thread return the boolean expression and sequential program, respectively.

Then, two typable controlled thread systems U and V satisfy the relation $S3$ if they have the same number of threads, the same scheduler P, which must be a guarded program, the same (low) guards for each thread and all the sequential threads satisfy one-to-one a relation, say S, that is a quasi-bisimulation. We prove that the corresponding relation on configurations R_3 is a strong bisimulation.

lemma *R3-is-csbis*: *csbis R*3

There is still one last relation:

constdefs $S4 :: (control \times control)\ set$
$S4 \equiv \{(U,\ V).\ (cship\ U \wedge cship\ V) \vee U\ S_3\ V\}$

The non-interference result relies on the following two lemmas. The first one establishes that the relation R_4 between configurations of controlled thread systems is a quasi-strong bisimulation.

lemma *R4-is-cqsbis*: *cqsbis R*4

The proof is easy using the previous two results. The second lemma connects typable programs and the relation S_4.

lemma *S4-refl*: *ctypable* $U \Longrightarrow U\ S_4\ U$

We are now ready to prove the main result.

theorem *ScheduledThreadSystems-Noninterference*:
ctypable $U \Longrightarrow$ *csecure* U

By instantiating with $S4$ the only tricky step that remains is the implication *cqsbis* $R \Longrightarrow$ *cbis* R. This is not true in general. However, it holds for any relation R satisfying the following condition:

$$\forall P\ Q\ \mu\ \nu.\ ship\ P \longrightarrow ship\ Q \longrightarrow \mu \simeq \nu \longrightarrow ((P,\ \mu),\ (Q,\ \nu)) \in R$$

which is easily proven for R_4.

5. CONCLUSION

We have presented what we believe to be the first machine-checked proof of non-interference for a concurrent language inspired from [4], and featuring primitives for scheduling. By using a proof assistant, we were able to eliminate several minor flaws from [4], and to extend the scope of the results of [4] by lifting convenient, but inessential restrictions, on the syntax of programs. Our work demonstrates that it is advantageous to use proof assistants in the design or the verification of advanced type systems for programming languages.

In the future, our work can be pursued in several directions.

Language expressiveness. We would like to generalize our results by extending the scope of the language, or considering variations of it. First of all, we would like to investigate when one can allow `when` and control expressions arbitrarily in programs, or equivalently to collapse processes, threads, and controlled thread systems into a single inductive definition. We would also like to extend the programming language with procedures, as done e.g. in [29], and with an exception handling mechanism, as done e.g. in [20].

Second of all, we would like to consider reactive programming, as studied by Matos, Boudol, and Castellani [15]. The reactive language they study includes such features as broadcast signals, suspension, preemption, and instants. However, it remains reasonably

constdefs qsbis :: (((seq × memory) × (seq × memory)) set) ⇒ bool
qsbis R ≡ sym R
∧ (∀ P μ Q ν. ((P, μ), (Q, ν)) ∈ R ⟶ μ ≃ ν
∧ ((∀ P′ μ′. (P, μ) ⟶$_s$ (P′, μ′) ⟶ (∃ Q′ ν′. (Q, ν) ⟶$_s$ (Q′, ν′) ∧ ((P′, μ′), (Q′, ν′)) ∈ R))
∨ (sship P ∧ sship Q)))

Figure 7: Quasi-bisimulation for sequential programs.

constdefs csbis :: (((control × memory) × (control × memory)) set) ⇒ bool
csbis R ≡ sym R
∧ (∀ P μ Q ν. ((P, μ), (Q, ν)) ∈ R ⟶ μ ≃ ν
∧ (∀ P′ μ′. (P, μ) ⟶$_c$ (P′, μ′) ⟶ (∃ Q′ ν′. (Q, ν) ⟶$_c$ (Q′, ν′) ∧ ((P′, μ′), (Q′, ν′)) ∈ R)))

Figure 8: Strong bisimulation for control programs.

close to the language presented here, and it should be possible to adapt our formal proofs.

More generally, it could be of interest calculi of mobile processes, such as the π-calculus, for which Isabelle formalizations exist, see e.g. [21]. One could use these existing formalizations as a basis to formally machine-check the correctness of information flow type systems for the π-calculus, such as those discussed in the introduction.

Security policy. Bisimulation-based definitions of security, such as the one adopted in this paper, are often too restrictive in practice. Finding more liberal yet meaningful definitions of security in a concurrent context is a challenging avenue of research, and it would be interesting to use our formalization as a basis for exploring more relaxed type systems that enforce weaker notions of security.

Yet another important research challenge currently being addressed by the language-based security community is the design of security definitions and of type systems that allow a controlled form of information release. Our objective here is to machine-check recent results in this area, e.g. results about downgrading and intransitive noninterference [14], delimited information release [24] and robust declassification [16, 31, 33].

Type-preserving compilation. We are currently working of machine-checked proofs of non-interference for the low-level language of [3]. It would be interesting to use the formalisation of this paper to give a machine-checked proof of correctness for the type preserving compiler described in that paper. We believe that existing experience with formalizing type-preserving compilers, see e.g. [13] will prove useful here. (In the formalization work for [3], we are using the proof assistant Coq [5], but most of the proofs of this paper have also been developed in that system.)

Acknowledgments. Thanks to the anonymous referees for their comments on the paper, and to G. Boudol, I. Castellani, and A. Matos for discussions around [4] and [15]. This work was partially funded by the IST Project Profundis, and by the ACI Sécurité SPOPS.

6. REFERENCES

[1] A. Banerjee and D. Naumann. Stack-based access control for secure information flow, 2003. Submitted for journal publication.

[2] A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.

[3] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In B. Steffen and G. Levi, editors, *Proceedings of VMCAI'04*, volume 2934 of *Lecture Notes in Computer Science*, pages 2–15. Springer-Verlag, 2004.

[4] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002. Preliminary version available as INRIA Research report 4254.

[5] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.4*, February 2003.

[6] S. Crafa, M. Bugliesi, and G. Castagna. Information flow security for boxed ambients. In V. Sassone, editor, *Proceedings of F-WAN*, volume 66(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2002.

[7] R. Focardi and R. Gorrieri. Classification of security properties: (part i: Information flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 331–396. Springer-Verlag, 2001.

[8] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.

[9] M. Hennessy and J. Riely. Information flow vs. resource access in the information asynchronous pi-calculus. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Proceedings of ICALP'00*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer, 2000.

[10] T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In R. Alur and I. Lee, editors, *Proceedings of EMSOFT'03*, volume 2855 of *Lecture Notes in Computer Science*, pages 241 – 256, 2003.

[11] K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proceedings of*

```
constdefs S3 :: (control × control) set
  S3 ≡{(U, V). ∃ P T T′. U=P⟦T⟧ ∧ V=P⟦T′⟧ ∧ length T=length T′
       ∧ ctypable U ∧ ctypable V ∧ guarded P
       ∧ (∀i<length T. guard-of-when (T!i) = guard-of-when (T′!i)
       ∧ L (secb( guard-of-when (T!i)))
       ∧ (∃ S. (seq-of-when (T!i), seq-of-when (T′!i)) ∈ S
       ∧ qsbis {((P, μ), (Q, ν)). (P, Q) ∈ S ∧ μ ≃ ν}) )}
```

Figure 9: Relation S_3.

POPL'02, pages 81–92. ACM Press, 2002.

[12] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.

[13] G. Klein and M. Strecker. Verified Bytecode Verification and Type-Certifying Compilation. *Journal of Logic and Algebraic Programming*, 58:27–60, 2004.

[14] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of APLAS'04*, Lecture Notes in Computer Science. Springer-Verlag, 2004. To appear.

[15] A. Almeida Matos, G. Boudol, and I. Castellani. Typing noninterference for reactive programs. In A. Sabelfeld, editor, *Proceedings of FCS'04*, 2004.

[16] A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proceedings of CSFW'04*, pages 172–186. IEEE Press, 2004.

[17] D. Naumann. Machine-checked correctness of a secure information flow analyzer (preliminary report). Technical Report CS-2004-10, Stevens Institute of Technology, March 2003.

[18] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[19] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.

[20] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of POPL'02*, pages 319–330. ACM Press, 2002.

[21] C. Röckl and D. Hirschkoff. A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *Journal of Functional Programming*, 2003.

[22] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-02, Computer Science Laboratory, SRI International, dec 1992.

[23] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.

[24] A. Sabelfeld and A. Myers. A model for delimited information release. In *Proceedings of ISSS'03*, Lecture Notes in Computer Science. Springer-Verlag, 2004. To appear.

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

[26] G. Smith. A New Type System for Secure Information Flow. In *Proceedings of CSFW'01*, pages 115–125, 2001.

[27] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL'98*, pages 355–364. ACM Press, 1998.

[28] M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[29] D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.

[30] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7:231–253, November 1999.

[31] S. Zdancewic and A.C. Myers. Robust declassification. In *Proceedings of CSFW'01*, pages 15–23. IEEE Press, 2001.

[32] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Proceedings of CSFW'03*. IEEE Press, 2003.

[33] S. Zwandewic. A type system for robust declassification. In S. Brookes and P. Panangaden, editors, *Proceedings of MFPS'03*, volume 83 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2003.