

Noninterference with Dynamic Security Domains and Policies

Robert Grabowski and Lennart Beringer

Ludwig-Maximilians-Universität
D-80538 München, Germany
{robert.grabowski,lennart.beringer}@ifi.lmu.de

Abstract. Language-based information flow analysis is used to statically examine a program for information flows between objects of different security domains, and to verify these flows follow a given policy.

When the program is distributed as mobile code, it may access resources whose domains depend on the client environment, or may face different security policies. In proof-carrying code scenarios, it is desirable to give a single proof that the program executes securely in any of these situations. This paper presents an object-oriented, Java-like language with runtime security types that can be inspected to ensure that flows between accessed objects are actually allowed before operations inducing these flows are performed. A type system is used to statically prove that the flow tests included in the program are sufficient, such that a noninterference property for the program is ensured regardless of the domains of objects and the effective security policy. Also, the paper outlines how the concepts of the type system are transferred to a bytecode language.

1 Introduction

The goal of information flow security is to control the flow of information between data objects of a computing system, like variables, files, or sockets. More precisely, each object is assigned a *security domain*. An *information flow policy* defines the allowed flows between objects of these domains. Programs that transfer data between objects in a manner that respects a transitive end-to-end flow policy are called *noninterferent* [1]. A way to verify this property is to perform a static analysis prior to the execution [2].

While the language-based approach has evolved into a larger research field [3], most works assume the objects a program uses and their domains can be statically inferred. In mobile code scenarios, however, the program is executed in different client environments, where the set of available data objects and their domains as well as the security policy typically vary.

For example, consider a program that opens a specific file and appends some data to it. The security of the program clearly depends on the domain of the contents of the accessed file and of the appended data as well as on the information flow policy. This makes it impossible to certify programs with current proof-carrying code (PCC) techniques [4], as the code producer has no or little

information about the environments the program is executed in, and therefore cannot prove security statically.

Another application scenario is an address book with entries whose fields have varying, user-defined confidentiality levels. A backup application should preserve these levels and the security guarantee in the backup copy, while a program that synchronizes the data with some less trusted cloud storage service should only transmit those contact profile fields that have been declared public. In general, a program should be able to adapt to the security environment.

A solution is to introduce language support for querying or reflecting the domains of objects and the security policy, such that potentially insecure operations can be guarded with an appropriate flow test.

Previous work The JIF language [5] contains constructs for dynamic label tests which have been formalized in the functional language λ_{DSec} [6]. Security domains can be used as values, enabling a constrained form of dependent pair and function types. In the imperative language RTI [7], data is associated with roles (sets of principals) which may be updated and queried programmatically to ensure that data flows securely. In the functional language λ^{deps^+} [8], all values are explicitly paired with their security domains, such that an external monitor program may throw invalid flow exceptions as required.

While all three approaches tackle aspects of dynamic information flow security in expressive and sophisticated ways, they are in our opinion not directly applicable to the mobile code scenario, as their analyses depend on a known security policy. Also, the analyses are defined only on the source code, and there is no translation given for an analysis on a lower level, e.g. bytecode.

The DSD language In this paper, we present a Java-like object-oriented language called DSD (for “Dynamic Security Domains”). It features a light-weight extension in form of abstract domains and flow operators that can be used to query dynamic domains and the security policy at runtime. The analysis does not make any assumptions about the available set of security domains or the flow policy by reasoning over the domains abstractly. With a single proof, programs can be shown secure for *any* object domains and security policies.

Moreover, the choice of a mild extension of Java makes it possible to transfer the analysis to the bytecode level by building on existing work on certified compilation that preserves security types [9–11]. The mechanism gives an end-to-end noninterference guarantee for compiled code suitable for PCC contexts.

In the language, we allow a very restricted form of dependent types for objects: each object has a special field f_δ that can be used as the symbolic domain of other fields. Likewise, we assume each method has a special variable x_δ that can be used as a type for the other variables.

For the file append example above, a file with a dynamic domain could be modelled as an object of a class `File` whose pseudo-specification is shown in figure 1. The *value* of the field f_δ is used as the runtime security domain of the field *contents*. Existing files and their security domains as given by the environment can then be modelled as objects initially present on the heap.

```

class File {
  fδ: Domain⊥;
  contents: Stringfδ;

  append(xδ: Domain⊥, s: Stringxδ) {
    if xδ ⊆ this.fδ then this.contents := this.contents + s;
    else skip; // or some error-handling code
  }
}

```

Fig. 1. File class

The method *append* takes a string s whose security type is explicitly passed in an additional domain argument x_δ . Since appending s to the file contents induces a flow of information, the contents update is guarded by a test whether the domain of the string (x_δ) is lower or equal to the domain of the file contents ($this.f_\delta$) with respect to the effective security policy. The method is secure, since the critical assignment to $this.contents$ is only executed if the induced flow is permitted in the client environment. The well-typedness of the methods proves that the file operations act securely in any particular client environment.

Outline of the analysis We present a type system in the style of Volpano, Smith, and Irvine [12] and Banerjee and Naumann [13]. Since the domains are not statically available, the analysis is performed by collecting symbolic information about the domain fields and variables. For instance, the expression $x_\delta \sqsubseteq this.f_\delta$ evaluates to true in the **then** branch of the *append* method. The system aims to derive as much as possible from such information by employing a technique similar to Hoare logic, and verifies the flow tests in the program are sufficient.

This work extends a previous paper [14], featuring a more standard object-oriented language with dynamic methods including domain variables and constraint annotations, and improves the analysis with flexible constraint sets, updatable domain variables and fields, and a more clarified abstract reasoning.

Main contributions This paper makes the following contributions:

- an object-oriented language with dynamic domains modelled explicitly as field and variables (section 2),
- security types that refer symbolically to these fields and variables, and a generalized termination-insensitive noninterference property (section 3), and
- a sound type system with abstract reasoning on the domain variable order (sections 4-5).

Furthermore, we briefly explain how the described dynamic security domains and concepts of the type system can be translated to the bytecode level (section 6). The presented type system for DSD has also been implemented as a prototypic type inference algorithm for a subset of Java with dynamic domain annotations [15]. For space reasons, this implementation is not discussed here.

2 The DSD Language

2.1 Syntax

The syntax of DSD is shown below. We use several disjoint sets of identifiers for variables ($x \in \mathcal{X}$), fields ($f \in \mathcal{F}$), classes ($C \in \mathcal{C}$), and methods ($m \in \mathcal{M}$). The notation \bar{e} denotes a sequence of argument expressions.

$$\begin{aligned} e \in \mathcal{E} &::= n \mid x \mid e.f \mid e \mathbf{op} e \mid \top \mid \perp \mid e \sqcup e \mid e \sqsubseteq e \\ P \in \mathcal{P} &::= P ; P \mid \mathbf{if} \ e \ \mathbf{then} \ P \ \mathbf{else} \ P \ \mathbf{while} \ e \ \mathbf{do} \ P \mid \\ &\quad \mathbf{skip} \mid x := e \mid e.f := e \mid x := \mathbf{new} \ C \mid x := e.m(\bar{e}) \end{aligned}$$

The syntax is split into expressions and programs (statements). It forms an imperative object-oriented language, with some additional expressions (\top , \perp , and operators \sqcup and \sqsubseteq) to refer abstractly to security domains, as well as arbitrary side-effect free operators (denoted by $e \mathbf{op} e$). The syntax does not depend on the concrete security domains and the policy, which are understood to be given by the environment in which the program is executed.

2.2 Domain lattice and semantics

The flow policy on the client is specified by a *domain lattice* $(\mathcal{D}, \leq, \vee, H, L)$. Given two security domains $k_1, k_2 \in \mathcal{D}$, the order $k_1 \leq k_2$ expresses that information may flow from k_1 to k_2 . To simplify presentation, we call the top-most domain in the lattice H (highest security) and the bottom-most domain L (lowest security), and assume a fixed domain lattice \mathcal{D} for now.

We use a standard object-oriented state model. It consists of a store, i.e. a variable valuation, and a heap where objects are allocated:

$$\begin{array}{ll} \text{states:} & \sigma \in \Sigma = \mathcal{S} \times \mathcal{H} & \text{values: } v \in \mathcal{V} ::= n \mid k \mid a \mid \text{null} \\ \text{stores:} & s \in \mathcal{S} = \mathcal{X} \rightarrow \mathcal{V} & \text{addresses: } a \in \mathcal{A} \\ \text{heaps:} & h \in \mathcal{H} = \mathcal{A} \rightarrow \mathcal{O} & \text{domains: } k \in \mathcal{D} \\ \text{objects: } & (C, F) \in \mathcal{O} = \mathcal{C} \times (\mathcal{F} \rightarrow \mathcal{V}) & \text{numbers: } n \in \mathbb{N} \end{array}$$

There is a notable extension to the standard model: It is assumed there is a special variable identifier $x_\delta \in \mathcal{X}$, and a special field identifier $f_\delta \in \mathcal{F}$. We additionally require that a well-formed store s must contain x_δ , and all objects on a well-formed heap h must include the field f_δ . These assumptions are merely to simplify the presentation of type environments, which may refer to these special variables or fields. The analysis could later be extended to objects with none or many of these fields with arbitrary names.

Given a domain lattice \mathcal{D} , the interpretation of expressions in a program state σ is defined by a denotational semantics $\llbracket e \rrbracket_\sigma$. In particular, abstract domain expressions are interpreted in the concrete domain lattice:

$$\begin{aligned} \llbracket \top \rrbracket_\sigma &= H & \llbracket e_1 \sqcup e_2 \rrbracket_\sigma &= \llbracket e_1 \rrbracket_\sigma \vee \llbracket e_2 \rrbracket_\sigma \\ \llbracket \perp \rrbracket_\sigma &= L & \llbracket e_1 \sqsubseteq e_2 \rrbracket_\sigma &= \llbracket e_1 \rrbracket_\sigma \leq \llbracket e_2 \rrbracket_\sigma \end{aligned}$$

For programs, we use a big-step operational semantics $\sigma_1 \xrightarrow{P} \sigma_2$ which states that if P is executed in state σ_1 , it terminates in state σ_2 . The definition is completely standard, with the exception that $x := \mathbf{new} C$ initializes the value of the f_δ field of the newly created object with the domain L . For reasons of limited space, the full semantics is not given here.

For the dynamic method dispatch, we use the functions `margs` and `mtable` that describe for each method the names of the formal arguments as well as the (dynamic-class dependent) implementations:

$$\text{margs} : \mathcal{M} \rightarrow \overline{\mathcal{X}} \quad \text{mtable} : \mathcal{C} \times \mathcal{M} \rightarrow \mathcal{P}$$

Again to simplify the presentation, we require that each method includes x_δ at the first argument position. Methods are called dynamically depending on the dynamic type of the called object. A method is executed with a new store consisting of the formal parameters `margs(m)` initially bound to the actual argument values, a special variable *this* containing the caller object, as well as a special return variable *ret* whose contents is assigned to the variable x after the method has terminated. With *ret*, no special return syntax is needed.

3 Noninterference

Our type system employs variable and field type environments. The types focus on information flow security, hence we consider only programs that are well-typed with respect to data types.

3.1 Type environments

Type environments assign symbolic security domains to variables and fields:

$$\begin{aligned} \Gamma : \mathcal{X} &\rightarrow \{\top, \perp, x_\delta\} \\ \text{ft} : \mathcal{F} &\rightarrow \{\top, \perp, f_\delta\} \end{aligned}$$

A variable typing Γ associates a symbolic security domain with the variables of the active method body. The types \top or \perp refer abstractly to the top-most and bottom-most domain of \mathcal{D} , respectively. A variable typed with the special symbol x_δ has the domain that is stored in the variable x_δ at runtime. A variable typing Γ is well-formed if $x_\delta \in \text{dom}(\Gamma)$ and $\Gamma(x_\delta) = \perp$.

A field typing `ft` associates a type with each field of an object. A field typed with f_δ has the domain that is stored in the field f_δ of the same object. A field typing `ft` is well-formed if `ft(fδ) = ⊥`. Issues of subtyping, inheritance, and well-formedness are handled uniformly by giving a type for each field *identifier*. In the following, we assume a fixed well-formed field typing `ft` and leave it implicit. (Other types for $\Gamma(x_\delta)$ and `ft(fδ)` will be discussed later.)

3.2 Type interpretation

Types of variables and fields are interpreted as security domains in \mathcal{D} , given stores s and field valuations F , respectively:

$$\begin{aligned} \llbracket \top \rrbracket_s &= H & \llbracket \top \rrbracket_F &= H \\ \llbracket \perp \rrbracket_s &= L & \llbracket \perp \rrbracket_F &= L \\ \llbracket x_\delta \rrbracket_s &= s(x_\delta) & \llbracket f_\delta \rrbracket_F &= F(f_\delta) \end{aligned}$$

The interpretation is well-defined, as every store includes x_δ and every object has a field f_δ . If $\llbracket \Gamma(x) \rrbracket_s \leq k$, we say x is *visible at k* in s ; similar for fields.

3.3 Equivalence of states

In preparation for the definition of noninterference, we define when two states are equivalent with respect to a security domain $k \in \mathcal{D}$. To capture related allocations of different fresh addresses in two parallel runs, we parametrize equivalence by a partial bijection β , as presented in [13]. Two addresses a, a' are *indistinguishable* if $\beta(a) = a'$. Two domains, numbers, or null values are indistinguishable if they are equal. We write $v \sim_\beta v'$ for indistinguishable values.

For $k \in \mathcal{D}$, two stores are *k -equivalent* with respect to a variable typing Γ if all variables visible at k have indistinguishable values:

$$s \sim_\beta^{\Gamma, k} s' \iff \forall x \in \text{dom}(\Gamma). \llbracket \Gamma(x) \rrbracket_s \leq k \Rightarrow s(x) \sim_\beta s'(x)$$

Note that the relation is symmetric because $\llbracket \Gamma(x) \rrbracket_s = \llbracket \Gamma(x) \rrbracket_{s'}$: the well-formedness condition $\Gamma(x_\delta) = \perp$ ensures that x_δ itself contains the same domain in both states, therefore $\Gamma(x)$ is always interpreted as the same domain.

Two heaps are *k -equivalent* if all β -related objects are *k -equivalent*. Two objects are *k -equivalent* if they have the same fields and if all fields visible at k have indistinguishable values. In particular, this means that $F(f_\delta) = F'(f_\delta)$.

$$\begin{aligned} h \sim_\beta^k h' &\iff \text{dom}(\beta) \subseteq \text{dom}(h) \wedge \text{rng}(\beta) \subseteq \text{dom}(h') \wedge \\ &\quad \forall a \in \text{dom}(\beta). h(a) \sim_\beta^k h(\beta(a)) \\ (C, F) \sim_\beta^k (D, F') &\iff C = D \wedge \forall f \in \text{dom}(F). \llbracket \text{ft}(f) \rrbracket_F \leq k \Rightarrow F(f) \sim_\beta F'(f) \end{aligned}$$

Finally, we extend *k -equivalence* point-wise to program states:

$$(s, h) \sim_\beta^{\Gamma, k} (s', h') \iff s \sim_\beta^{\Gamma, k} s' \wedge h \sim_\beta^k h'$$

3.4 Noninterference

Using the definition of state equivalence, we now define information flow security for a program P as a standard termination-insensitive noninterference property.

Definition 1. P is secure with respect to variable typing Γ if for all domains $k \in \mathcal{D}$, for all states $\sigma_1, \sigma_2, \sigma'_1, \sigma'_2$ and for all partial bijections β ,

$$\sigma_1 \sim_{\beta}^{\Gamma, k} \sigma'_1 \wedge \sigma_1 \xrightarrow{P} \sigma_2 \wedge \sigma'_1 \xrightarrow{P} \sigma'_2 \Rightarrow \sigma_2 \sim_{\gamma}^{\Gamma, k} \sigma'_2$$

for some partial bijection $\gamma \supseteq \beta$.

The property implies that all objects that are β -related before the execution stay related after the execution. The extended bijection γ captures the fact that new addresses may have been allocated and related in both executions.

As noninterference with dynamic security domains may rely on the values of domain fields and variables, we parametrize the above definition by certain state predicates Q and Q' that hold before and after the execution. We write $\sigma \models Q$ if σ satisfies Q , and will instantiate these predicates in the next section.

Definition 2. Let Q and Q' be state predicates, and P a program. P is (Q, Q') -valid if for all states σ_1 and σ_2 such that $\sigma_1 \xrightarrow{P} \sigma_2$, $\sigma_1 \models Q$ implies $\sigma_2 \models Q'$. P is (Q, Q') -secure with respect to Γ if it is secure w.r.t. Γ and (Q, Q') -valid.

4 Reasoning with Abstract Security Domains

4.1 Labels

As x_{δ} and f_{δ} may occur symbolically in variable and field types, the type system infers abstract domains as types for expressions. These abstract domains are called *labels*, and are a subset of expressions:

$$\begin{aligned} \text{access paths:} \quad & \pi ::= x \mid \pi.f \\ \text{labels:} \quad & \mathcal{L} \ni \ell ::= \top \mid \perp \mid x_{\delta} \mid \pi.f_{\delta} \mid \ell \sqcup \ell \end{aligned}$$

An f_{δ} field can appear in a label if the object to which it belongs is referenced in a normalized form, i.e. by an access path π of the form $x.f_1.f_2.\dots.f_n$. This restriction is needed since objects are used syntactically in labels.

The type system assigns a label to each expression. For example, if $\Gamma(y) = x_{\delta}$ and $\Gamma(z) = \top$, the expression $y + z$ is assigned the label $x_{\delta} \sqcup \top$. Every label is an expression that evaluates to a security domain. If an expression e is typed with a label ℓ , then e depends in state σ on data of domains at most $\llbracket \ell \rrbracket_{\sigma}$. Since labels are special expressions, DSD features a very constrained form of dependent types.

4.2 Ordering labels

Information flow type systems with static domains contain a number of side conditions $k_1 \leq k_2$ which require that information may flow from security domains k_1 to k_2 according to the security policy. Our type system has no static information about the values of labels at runtime. It is nevertheless possible to define an abstract order on labels by exploiting the fact that elements in \mathcal{D} (to

which labels evaluate) are ordered as a lattice. For example, it can be inferred that data may always flow from an expression labeled with \perp to an expression labeled with $x_\delta \sqcup y.f_\delta$, as the evaluation of \perp is always the lowest element in \mathcal{D} .

The following rules define the order over labels. (Assume $Q = \emptyset$ for now.) We additionally define a label equality \equiv mirroring idempotence, commutativity, and associativity of the join operator, as well as antisymmetry of the order.

$$\frac{}{\perp \sqsubseteq_Q \ell} \quad \frac{}{\ell \sqsubseteq_Q \top} \quad \frac{}{\ell \sqsubseteq_Q \ell \sqcup \ell'} \quad \frac{\ell_1 \sqsubseteq_Q \ell_2 \quad \ell_2 \sqsubseteq_Q \ell_3}{\ell_1 \sqsubseteq_Q \ell_3} \quad \frac{\ell_1 \sqsubseteq_Q \ell_3 \quad \ell_2 \sqsubseteq_Q \ell_3}{\ell_1 \sqcup \ell_2 \sqsubseteq_Q \ell_3} \quad \frac{(\ell_1, \ell_2) \in Q}{\ell_1 \sqsubseteq_Q \ell_2}$$

The label order rules that are justified by properties of the domain lattice are usually not sufficient to typecheck a program, as the validity of flows may depend on the actual values of domain fields and variables. This is the reason why the language features label flow tests, i.e. conditional statements of the form **if** $\ell_1 \sqsubseteq \ell_2$ **then** P_1 **else** P_2 . For the subprogram P_1 , we can assume that an information flow from $[[\ell_1]]_\sigma$ to $[[\ell_2]]_\sigma$ is permitted; otherwise, the branch will not be taken during the execution. The typing judgements for programs are therefore parametrized over a set $Q \subseteq \mathcal{L} \times \mathcal{L}$ containing label pairs. A pair $(\ell_1, \ell_2) \in Q$ expresses the assumption that a flow from ℓ_1 to ℓ_2 is allowed.

The set Q thus stores abstract information about f_δ fields and x_δ variables at a point of execution. Since Q gives requirements for suitable program states, we also call it the *constraint set*. (Constraint sets are the state predicates that were used in the definition of noninterference.)

Definition 3. A program state σ satisfies a constraint set Q , written $\sigma \models Q$, if for all pairs $(\ell_1, \ell_2) \in Q$ it holds that $[[\ell_1]]_\sigma \leq [[\ell_2]]_\sigma$.

The following theorem states that the rules for label order and equality are sound with respect to their interpretation in satisfying program states.

Theorem 1 (Soundness of label order and equality rules). Given a set of constraints Q , two labels ℓ and ℓ' and a state σ satisfying Q , then

1. $\ell_1 \sqsubseteq_Q \ell_2$ implies $[[\ell_1]]_\sigma \leq [[\ell_2]]_\sigma$, and
2. $\ell_1 \equiv \ell_2$ implies $[[\ell_1]]_\sigma = [[\ell_2]]_\sigma$.

Proof. By induction over the derivation of the label order and equality.

In fact, $(\mathcal{L}, \sqsubseteq_Q, \sqcup, \top, \perp)$ forms a semi-lattice over the (infinite) set of labels, where labels related by \equiv refer to the same lattice point. It can be shown that if σ satisfies Q , the evaluation function $[[\cdot]]_\sigma$ is a homomorphism that embeds the label lattice into the domain lattice.

The evaluation function not only depends on the program state σ , but also on \mathcal{D} and its lattice operators. If we fix the state σ , then Q can as well be interpreted as the set of those domain lattices (\mathcal{D}, \leq) whose structure includes the pairwise domain positionings which are abstractly described by label pairs in Q . A label test thus collects information about the program state and the structure of the domain policy at once.

5 Type System

The type system is mostly syntax-directed, and follows the separation of expressions and programs. We mainly discuss the extensions and differences to the type systems by Volpano, Smith, and Irvine [12] and Banerjee and Naumann [13].

5.1 Typing expressions

The typing rules for expressions are shown in figure 2. A typing judgement $\Gamma \vdash e : \ell$ means that the expression e has a label ℓ , i.e. it depends in a specific program state σ on information of security domain $\llbracket \ell \rrbracket_\sigma$ at most. In particular, the type of a variable x , which is looked up in the environment Γ , is from the set $\{\top, \perp, x_\delta\}$ and thus a label. For field access, we define the *qualified field type*

$$\text{ft}_\pi(f) := \begin{cases} \pi.f_\delta & \text{if } \text{ft}(f) = f_\delta \\ \text{ft}(f) & \text{else} \end{cases}$$

to transform the field type f_δ into a label by prepending an access path π that is supposed to reference the object whose field f is accessed.

5.2 Typing programs

For programs, we derive a typing judgement $\Gamma, pc \vdash Q \{P\} Q'$, which means a program P is secure if executed in states satisfying Q , and finishes in states satisfying Q' . The rules for the derivation system are shown in figure 3. Basically, the rules combine a Hoare logic-style reasoning on program predicates (constraint sets) with an information flow type system that uses labels instead of domains.

As in [13], the *program counter label* pc is actually a pair of labels (pc_s, pc_h) , which is used to capture the lower bound of side effects on the store and the heap, respectively, in order to prevent indirect information flows. Using a pair of labels improves precision when method calls are involved.

The interesting rules are the ones for label tests and assignments. The rule for label tests works like the ordinary rule for conditionals, but also adds the label comparison to the pre-set of the **then** branch, since the condition holds when the execution takes that branch. No negative label order information is added for the **else** branch, since it does not improve the precision of the type system, and also might introduce self-contradicting constraint sets.

A variable assignment is typable if the induced flows can be abstractly shown secure, i.e. if the flows between labels are justified by the pre-condition set Q . It is also possible to update the x_δ variable, but only if it can be inferred that the new domain, given by the expression e , is at least as high as the old one. Therefore, one can change the runtime type of data e.g. from L to H .

The set Q' is used for backward reasoning, in order to be able to make assertions about the post-value of x . The reason why Q is added is that one might need the pre-state of x in the premises of the rule. Note that $Q'[e/x]$ and Q do not need to be disjoint; in fact, they may even be identical. The constraint

$$\begin{array}{c}
\frac{}{\Gamma \vdash n : \perp} \quad \frac{c \in \{\top, \perp\}}{\Gamma \vdash c : \perp} \quad \frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma \vdash \pi : \ell}{\Gamma \vdash \pi.f : \text{ft}_\pi(f) \sqcup \ell} \\
\frac{\circ \in \{\mathbf{op}, \sqcup, \sqsubseteq\} \quad \Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \circ e_2 : \ell_1 \sqcup \ell_2}
\end{array}$$

Fig. 2. Expression type system

$$\begin{array}{c}
\frac{\Gamma, pc \vdash Q_0 \{P\} Q'_0 \quad Q \Rightarrow Q_0 \quad Q'_0 \Rightarrow Q'}{\Gamma, pc \vdash Q \{P\} Q'} \\
\frac{\Gamma, pc \vdash Q \{P_1\} Q' \quad \Gamma, pc \vdash Q' \{P_2\} Q''}{\Gamma, pc \vdash Q \{\mathbf{skip}\} Q} \quad \frac{\Gamma \vdash e : \ell \quad \Gamma, pc \sqcup \ell \vdash Q \{P_1\} Q' \quad \Gamma, pc \sqcup \ell \vdash Q \{P_2\} Q'}{\Gamma, pc \vdash Q \{\mathbf{if } e \mathbf{ then } P_1 \mathbf{ else } P_2\} Q'} \\
\frac{\Gamma \vdash \pi : \ell_1 \quad \Gamma \vdash e : \ell_2 \quad \Gamma_1 \vdash \ell_1 \sqsubseteq \ell_2 : \ell \quad \Gamma, pc \sqcup \ell \vdash Q, \ell_1 \sqsubseteq \ell_2 \{P_1\} Q' \quad \Gamma, pc \sqcup \ell \vdash Q \{P_2\} Q'}{\Gamma, pc \vdash Q \{\mathbf{if } \ell_1 \sqsubseteq \ell_2 \mathbf{ then } P_1 \mathbf{ else } P_2\} Q'} \\
\frac{\Gamma \vdash e : \ell \quad \ell \sqcup pc_s \sqsubseteq_Q \Gamma(x) \quad x = x_\delta \Rightarrow x \sqsubseteq_Q e \quad x \notin pc}{\Gamma, pc \vdash Q'[e/x] \cup Q \{x := e\} Q'} \quad \frac{\Gamma \vdash \pi : \ell_1 \quad \Gamma \vdash e : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqcup pc_h \sqsubseteq_Q \text{ft}_\pi(f) \quad f = f_\delta \Rightarrow \pi.f \sqsubseteq_Q e \quad f \notin pc \quad f \notin Q'[e/\pi.f]}{\Gamma, pc \vdash Q'[e/\pi.f] \cup Q \{\pi.f := e\} Q'} \\
\frac{pc_s \sqsubseteq_Q \Gamma(x) \quad x \notin pc \quad x \neq x_\delta \quad x \notin Q'}{\Gamma, pc \vdash Q' \cup Q \{x := \mathbf{new } C\} Q', x.f_\delta \leq \perp} \\
\frac{\Gamma \vdash \pi : \ell_{this} \quad \Gamma \vdash \bar{e} : \bar{\ell} \quad \text{mtype}(m)[\bar{e}_{\#1}/x_\delta] = \ell'_{this}, \bar{\ell}' \xrightarrow{pc'_h} \ell_{ret} \quad \ell_{this} \sqsubseteq_Q \ell'_{this} \quad \bar{\ell} \sqsubseteq_Q \bar{\ell}' \quad pc_h \sqcup \ell_{this} \sqsubseteq_Q pc'_h \quad \ell_{ret} \sqcup \ell_{this} \sqcup pc_s \sqsubseteq_\emptyset \Gamma(x) \quad \text{mreq}(m)[\bar{e}/\text{margs}(m)][\pi/\text{this}] = Q \quad \text{mens}(m)[x/\text{ret}] = Q' \quad x \notin pc \quad x \neq x_\delta}{\Gamma, pc \vdash Q \{x := \pi.m(\bar{e})\} Q'}
\end{array}$$

where

- $id \notin \ell$ if and only if the identifier id does not syntactically occur in ℓ
- $Q \Rightarrow Q'$ if and only if $\forall (\ell_1, \ell_2) \in Q'. \ell_1 \sqsubseteq_Q \ell_2$
- $pc \sqcup \ell = (pc_s \sqcup \ell, pc_h \sqcup \ell)$

Fig. 3. Program type system

set assertions can be derived in Hoare logic by combining the rules for assignment and consequence of the logic. The rule for assigning to fields works similarly. Side conditions of the form $x \notin pc$ ensure that the evaluation of pc remains invariant.

The rule for method calls relies on given method annotations. More precisely, the (*security*) *type of a method* is a tuple $(t_{this}, \bar{t}, t_h, t_{ret})$ that assigns variable types to formal arguments of a method, where t is a meta-variable ranging over the possible variable types $\{\top, \perp, x_\delta\}$. We denote this by writing

$$\text{mtype}(m) = t_{this}, \bar{t} \xrightarrow{t_h} t_{ret},$$

to be read as: in method m , *this* has type t_{this} , the arguments $\text{margs}(m)$ have types \bar{t} , *ret* has type t_{ret} , and no fields below t_h are updated.

In the rule for method calls, these variable types are turned into labels by substituting x_δ with the first argument expression in the sequence, which is supposed to contain the domain value for x_δ . It is then checked whether the passed arguments and *this* have labels lower than their formal labels, and whether the return value has a label lower than the assigned variable. Also, the lower bound of the caller's heap side effect must be lower than the method's lower bound.

We also use functions

$$\text{mreq}, \text{mens} : \mathcal{M}(m) \rightarrow \mathcal{L} \times \mathcal{L}$$

to annotate method declarations with required (pre-)constraint sets and ensured (post-)constraint sets. Required sets may refer to the local variables, and ensured sets may refer to *ret*. The appropriate variable substitutions are performed before they can be compared with the caller's constraint sets. We can now formulate that a method is well-typed if its declared type can actually be derived.

Definition 4. *Let m be a method with $\text{mtype}(m) = t_{this}, \bar{t} \xrightarrow{t_h} t_{ret}$ and constraint sets $\text{mreq}(m) = Q$ and $\text{mens}(m) = Q'$. Then, m is well-typed if for all implementations P of m , the judgment $\Gamma, (\perp, t_h) \vdash Q \{P\} Q'$ can be derived, where*

$$\Gamma = [\text{margs}(m) \mapsto \bar{t}][\text{this} \mapsto t_{this}][\text{ret} \mapsto t_{ret}].$$

5.3 Soundness

The following is the main soundness theorem.

Theorem 2. *If $\Gamma, (\perp, \perp) \vdash Q \{P\} Q'$ and all methods are well-typed, then P is (Q, Q') -secure with respect to Γ .*

The proof of the theorem is by induction over the operational semantics. The full proof can be found on the first author's homepage [16].

We observe that the theorem actually states security for any given domain lattice: The semantics, state equivalence and security notion are parametric in the lattice \mathcal{D} . However, the syntax, the type environments and the typing rules only refer to labels and do not depend on the concrete security policy. Therefore, if a program is typable, it is secure with respect to any given security policy (\mathcal{D}, \leq) . As motivated in the introduction, this enables a single verification of a program that is executed in different environments with varying security policies.

5.4 Meta-label monotonicity

The soundness proof relies on an intrinsic property of the type system: a security label ℓ derived as a type always interprets to a domain that is at least as confidential as the interpretation of the (meta-)label of ℓ .

Lemma 1. *If $\Gamma \vdash e : \ell$ and $\Gamma \vdash \ell : \ell'$, then $\llbracket \ell' \rrbracket_\sigma \leq \llbracket \ell \rrbracket_\sigma$ for any state σ .*

To put it in informal terms: the fact that something is public can never be itself a secret. The property seems to be common in languages where security types are accessible programmatically; RTI [7] for example has a similar requirement on security roles.

For precisely that reason, we chose \perp as the type for domain variables and fields. If we admitted for instance $\Gamma(x_\delta) = \top$, a consistent type system would need to ensure that every x_δ -typed variable actually gets a label that is at least \top , which is not very useful.

6 Dynamic Security Domains at the Bytecode Level

This section outlines how the described dynamic security domains can be expressed on the bytecode level, i.e. for an unstructured language that operates on a stack. Since the described concepts are directly re-used, the type system is mainly suitable for results of the compilation from DSD.

A bytecode program P is a mapping of program points i to instructions I with the following syntax:

$$\begin{aligned} lit &::= \top \mid \perp \mid n \\ op &::= \sqcup \mid \sqsubseteq \mid \dots \\ I &::= \text{push } lit \mid \text{pop} \mid \text{prim } op \mid \text{load } x \mid \text{store } x \mid \text{ifeq } i \mid \text{goto } i \mid \\ &\quad \text{new } C \mid \text{putfield } f \mid \text{getfield } f \mid \text{call } m \mid \text{return} \end{aligned}$$

Instructions are interpreted with respect to program states (i, s, h, ρ) where i is the program point, s and h are stores and heaps as before, and ρ is an operand stack, i.e. a list of values. The small-step operational semantics for each instruction is omitted here, but can be found e.g. in [10].

The type system is based on the system by Barthe et al. [11], but requires the derivation of some additional information. Figure 4 is meant to illustrate the following explanations; it shows the bytecode program that corresponds to the introductory example **if** $x_\delta \sqsubseteq \text{this}.f_\delta$ **then** P_1 **else** P_2 , and gives a possible derivation of the required auxiliary mappings, assuming $\Gamma(\text{this}) = \top$.

In contrast to the original type system, we use labels instead of domains again. Since labels remain a subset of expressions of the high-level DSD language, it is necessary to “disassemble” instructions that are used to construct labels. This is accomplished by deriving judgements of the form

$$i \vdash E \Rightarrow E'$$

i	$\mathbf{E}(i)$	$\mathbf{Q}(i)$	$\mathbf{S}(i)$	$\mathbf{pc}(i)$	$P(i)$
1	ϵ	\emptyset	ϵ	\perp	load x_δ
2	" x_δ "	\emptyset	\perp	\perp	load <i>this</i>
3	" <i>this</i> " :: " x_δ "	\emptyset	$\top :: \perp$	\perp	getfield f_δ
4	" $x_\delta.f_\delta$ " :: " x_δ "	\emptyset	$\top :: \perp$	\perp	prim \sqsubseteq
5	" $x_\delta \sqsubseteq x_\delta.f_\delta$ "	\emptyset	\top	\perp	ifeq 17
6	ϵ	\emptyset	ϵ	\top	(instructions of $P_2\dots$)
\vdots					
17	ϵ	$\{x_\delta \sqsubseteq x_\delta.f_\delta\}$	ϵ	\top	(instructions of $P_1\dots$)

Fig. 4. Type derivation of the compilation of **if** $x_\delta \sqsubseteq x_\delta.f_\delta$ **then** P_1 **else** P_2

where E and E' are sequences of (high-level language) expressions and thus abstractions of the stack values. The judgement means that the execution of the instruction at i in a state with a stack described by E leads to a new state where the stack is described by E' . The system induces a mapping \mathbf{E} that associates each program point with a list of expressions that describe the stack whenever that point is reached. (Instead of an expression, one may also describe a stack entry with a special symbol that indicates “don’t know”, hence \mathbf{E} always exists.)

Using the stack expressions, we can compute local pre- and post-constraint sets for each instruction by deriving judgements

$$\mathbf{E} \vdash i : Q \Rightarrow i' : Q'$$

which say that if a state satisfies Q and program point i' is reachable via the instruction $P(i)$, then the new state satisfies Q' . Together with a consequence rule, this gives a mapping \mathbf{Q} from each program point to a constraint set.

Finally, the main typing judgment has the form

$$\Gamma, \mathbf{Q}, \mathbf{E}, \mathbf{pc}, i \vdash S \Rightarrow S'$$

which models the small-step semantics abstractly by giving stack typings S and S' , which are sequences of labels describing the security of the stack values at position i and the following position. The component \mathbf{pc} is a mapping of instruction points to triples of labels (pc_s, pc_h, pc_ρ) that give the lower bound on side effects on the store, the heap, and the stack at each position. As is standard, whenever a “high” branching is performed, the program counter label is lifted for every instruction within the *control dependence region* [17] of the branching instruction.

If the rules induce a mapping \mathbf{S} of program points to stack typings, then the program is well-typed, and (according to the soundness property) also secure. The soundness proof shows that the rules define a weak bisimulation of two executions where instructions with a “high” program counter label are shown to have no visible effect.

7 Discussion

We presented an object-oriented language with runtime tests of security domains, which makes programs possible that safely access dynamically typed data. This section highlights further benefits of DSD and outlines open work.

Modular security design The encapsulation of data and security domains supports a modular software design. In the introductory example, the caller of *append* does not need to deal with security-related aspects of the file's contents if an error recovery mechanism for the **else** case is implemented. Alternatively, it is possible with the presented framework to pass the proof obligation $x_\delta \sqsubseteq \text{this}.f_\delta$ to the method caller by annotating the method accordingly.

Erasure and declassification in DSD As the domain fields and variables can be inspected and compared during execution, it is possible to perform various security-related operations. For example, a file can be classified as confidential simply by setting its f_δ field to H , or information can be *erased* by overwriting the *contents* field with a harmless value and then setting f_δ to L . The last scenario is secure but currently not typable.

When the contents is not overwritten, the above operation amounts to *declassification*. Though it is hard to find an appropriate weakening of the noninterference property, the explicit change of security types (f_δ) in DSD represents a novel syntactic way of declassification.

Type preserving compilation For PCC scenarios, it is desirable to compile DSD programs to bytecode such that the bytecode translation is typable with the bytecode type system if the original program was typable with the high-level system. The compilation may take the structure of the high-level program into account to compute the control dependence regions. Both the compilation and the type preservation proof are currently work in progress.

Implementation As mentioned in the introduction, we have encoded the DSD language into a subset of the Java language by using Java annotations for field types and method signatures. Furthermore, we have developed a prototypic implementation of a type inference as a plugin for the integrated development environment Eclipse [15].

Expressivity and complexity We plan to give a larger example program to better demonstrate the expressivity of the type system. Also, although the focus of PCC lies on efficient verifiability, it would be interesting to investigate the complexity of the analysis, or to give empirical benchmark results of the implementation.

Acknowledgements

This work was supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and by the DFG-funded project InfoZert, grant

number Be 3712/2-1. We would like to thank Gilles Barthe and Daniel Hedin (IMDEA Software Madrid) as well as Alexander Knapp, Florian Lasinger, and Martin Hofmann (LMU Munich) for their helpful input and comments.

References

1. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proceedings of the 1982 Symposium on Security and Privacy, IEEE Computer Society Press (1982) 11–20
2. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. *Commun. ACM* **20**(7) (1977) 504–513
3. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* – special issue on Formal Methods for Security **21**(1) (January 2003) 5 – 19
4. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages, ACM Press (1997) 106–119
5. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 1999), ACM Press (1999) 228–241
6. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. *Int. J. Inf. Secur.* **6**(2) (2007) 67–84
7. Bandhakavi, S., Winsborough, W., Winslett, M.: A trust management approach for flexible policy management in security-typed languages. In: Proceedings of 21st IEEE Computer Security Foundations Symposium, Los Alamitos, CA, USA, IEEE Computer Society (2008) 33–47
8. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium, Washington, DC, USA, IEEE Computer Society (2007) 203–217
9. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference java bytecode verifier. In Nicola, R.D., ed.: ESOP. Volume 4421 of Lecture Notes in Computer Science., Springer (2007) 125–140
10. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation. (2005) 103–112
11. Barthe, G., Rezk, T., Naumann, D.A.: Deriving an Information Flow Checker and Certifying Compiler for Java. In: S&P, IEEE Computer Society (2006) 230–242
12. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. *J. Computer Security* **4**(3) (1996) 167–187
13. Banerjee, A., Naumann, D.A.: Stack-based access control and secure information flow. *J. Funct. Program.* **15**(2) (2005) 131–177
14. Grabowski, R.: Noninterference for Mobile Code with Dynamic Security Domains. In: International Workshop on Proof-Carrying Code, Pittsburgh, USA (2008) Post-proceedings to appear.
15. Lasinger, F., Grabowski, R.: DSecCheck: Implementation of the DSD type system as an Eclipse plug-in. <http://www.tcs.ifl.lmu.de/~grabow/dsd> (2009)
16. Grabowski, R.: Proofs for the soundness of the DSD type system. <http://www.tcs.ifl.lmu.de/~grabow/dsd> (2009)
17. Ball, T.: What’s in a region? or computing control dependence regions in near-linear time for reducible control flow. *LOPLAS* **2**(1-4) (1993) 1–16