Modular and Constraint-based Information Flow Inference for an Object-oriented Language

Qi Sun$^{\star 1}$, Anindya Banerjee$^{\star\star 2}$, and David A. Naumann$^{\star\star\star 1}$

^{1} Stevens Institute of Technology, USA {sunq,naumann}@cs.stevens-tech.edu
^{2} Kansas State University, USA ab@cis.ksu.edu

Abstract. This paper addresses the problem of checking programs written in an object-oriented language to ensure that they satisfy the information flow policies, confidentiality and integrity. Policy is specified using security types. An algorithm that infers such security types in a modular manner is presented. The specification of the algorithm involves inference for libraries. Library classes and methods maybe parameterized by security levels. It is shown how modular inference is achieved in the presence of method inheritance and override. Soundness and completeness theorems for the inference algorithm are given.

1 Introduction

This paper addresses the problem of checking programs to ensure that they satisfy the information flow policies, confidentiality and integrity. Confidentiality, for example, is an important requirement in several security applications – by itself, or as a component of other security policies (e.g., authentication), or as a desirable property to enforce in security protocols [1]. In the last decade, impressive advances have been made in specifying static analyses for confidentiality for a variety of languages [14]. Information flow policy is expressed by labeling of input and output channels with levels, e.g., low, or public, (L) and high, or secret, (H) in a security lattice $(L \leq H)$. Many of these analyses are given in the style of a security type system that is shown to enforce a *noninterference* property [6]: a well-typed program does not leak secrets.

Previous work of Banerjee and Naumann provides a security type system and noninterference result for a class based object-oriented language with features including method inheritance/overriding, dynamic binding, dynamically allocated mutable objects, type casts and recursive types [3]. It is shown how several object-oriented features can be exploited as covert channels to leak secrets. Type checking in Banerjee and Naumann's security type system requires manually annotating all fields, method parameters and method signatures with security types.

* Supported in part by NSF grant CCR-0208984 and NJCST.
** Supported in part by NSF grants CCR-0296182 and CCR-0209205.
*** Supported in part by NSF grant CCR-0208984 and NJCST.

The primary focus of this paper is the *automatic inference* of security type annotations of well-typed programs. In this paper, we are not interested in full type inference, and assume that a well-typed program is given. There are several issues to confront. First, we demand inference of some, possibly all, security levels of *fields* in a class. This means that security types of fields will involve *level variables* and the same is true for method types where level variables will appear in types of method parameters and in the result type.

The second issue, a critical challenge for scalability, is achieving *modular* security type inference for class-based languages. A non-modular, whole-program inference, say, for the language in [3], would perform inference in the context of the entire class table; if method m in class A is called in the body of method n declared in class B, then the analysis of $B.n$ would also involve the analysis of $A.m$. Moreover, every use of $A.m$ in a method body would necessitate its analysis. Our insistence on modular inference led us to the following choices: code is split into library classes (for which inference has already been performed) and the current analysis unit (for which inference is currently taking place). Inference naturally produces polymorphic types; so it seemed appropriate to go beyond previous work [3] and make libraries polymorphic. To track information flow, e.g., via field updates, *constraints* on level variables are imposed in the method signature; thus library method signatures appear as constrained polymorphic types. To avoid undecidability of inference due to polymorphic recursion [8, 7], mutually recursive classes and methods in the current analysis unit are analyzed monomorphically. [1]

Because we are analyzing an object-oriented language, the third issue we confront is achieving modular inference in the presence of method inheritance and override. The current analysis unit can contain subclasses of a library class with some library methods overridden or inherited. To achieve modularity, we require that the signature of a library method is *invariant* with respect to subclassing. Getting the technical details correct is a formidable challenge and one which we have met. We provide some intuition on the problem presently.

The research reported in this paper is being carried out in the context of a tool, currently under development, that handles the above issues. The tool helps the programmer design a library interactively by inferring the signatures of new classes, together with a constraint set showing the constraints that level variables in the signature must obey. The security types of the new classes are inferred in the context of the existing library. The new code may inherit library methods – this causes the polymorphic signature of a library method to be instantiated at every use of the method, and the instantiated constraints will apply to the current context.

Handling method override is more subtle; modularity requires that the polymorphic type inferred for a library method must be satisfied by all its overriding methods. For an overriding method in a subclass, if the inference algorithm

[1] It is possible that because we are not doing full type inference, polymorphic recursion in this setting is decidable. But we do not yet have results either way.

generates constraints that are not implied by the constraints of the superclass method, then the unit must be rejected.

To cope with such a situation, there are a couple of approaches one may adopt. Because changing library code makes the inference process non-modular, one can change the code in the subclass, by relabeling field and parameter levels with ground constants in a sensible way, and re-run the tool to deliver the relaxed signatures. (This will be illustrated by an example in section 4.2). A more practical approach is that during library design, the designer may want to consider anticipated uses of those library methods that are expected to be overridden in subclasses. The inferred signature of such library methods – an extreme example being abstract methods with no implementation – may be too general; hence, the designer may want to make some of the field types and method signatures more specific. Then, there would be more of a possibility that the constraints in the method signatures of library methods will imply those in the signatures of the overriding methods. Thus the security type signature we assume for library methods is allowed to be an arbitrary one.

Contributions and overview. This paper tackles all of the issues above. Our previous work [3] did not cope with libraries. Here, we do; thus we provide a new security type system for the language with polymorphic classes and methods that guarantees noninterference. We provide an inference algorithm that, in the context of a polymorphic library, infers security type signatures for methods of the current analysis unit. By restricting the current analysis unit to only contain monomorphic types, we can show that the algorithm computes principal monomorphic types, as justified by the completeness of the inference algorithm for such restricted units. Although there have been studies about both type inference and information flow analysis for imperative, functional and object-oriented programs, we have not found any work that addresses security type inference for object-oriented programs in the presence of libraries. We believe that the additional details required to account for modularity in the presence of method inheritance and override are a novel aspect of our work.

After discussing two simple examples in section 2, we describe the language extended with parameterized classes in section 3, explain the inference algorithm in section 4 and give the soundness and completeness theorems in section 5. Related work and a discussion of the paper appear in section 6.

2 Examples

Consider the following classes:

```
class TAX extends Object {
  int income;
  int tax(int salary){ self.income := salary; result := self.income * 0.20; }}
class Inquiry extends Object{
  TAX employee; int est;
  bool overpay(){int tmp:=employee.tax(1000);result:=(tmp≥self.est);}}
```

The type of method *tax* in class *TAX*, written *mtype*(*tax*, *TAX*), is *int* → *int*. Assuming that *income* has security level H, a possible security type for method *tax*, is $L, H \multimap\!\langle H\rangle\!\!\to H$: when the level of the current object[2] is L and the level of *salary* is at most H then *tax* returns a result of level at most H and only H fields (the H in the middle) may be updated[3] during method execution. This security type can be verified using security type checking rules for method declaration and commands.

In security type inference, we *infer* security types for the level of field *income* and for method *tax*. We assume that *TAX* is a well-formed class declaration. The inference algorithm is given class *TAX* as input, but with *income*'s type annotated, e.g., as $(\mathbf{int}, \alpha_1)$, where α_1 is a placeholder for the actual level of the field. It is also possible for *income* to be annotated with a constant level (e.g., L or H). Apart from annotating fields, we also annotate the parameter and return types of methods. In the sequel, we will let letters from the beginning of the Greek alphabet range over level variables.

The type, $(\alpha_2, \alpha_5 \multimap\!\langle \alpha_3\rangle\!\!\to \alpha_4) \mid \{\alpha_5 \le \alpha_1, \alpha_3 \le \alpha_1, \alpha_2 \le \alpha_1, \alpha_1 \le \alpha_4\}$ is inferred for method *tax*; that is, if the level of the *TAX* object is at most α_2 and the level of *salary* is at most α_5, then the level of the return result is at most α_4 and fields of level at most α_3 can be assigned to during execution of *tax*. The first constraint precludes, e.g., assigning H value to L field.

The class *TAX* can now be converted into a library class, parameterized over α_1 and method *tax* given a polymorphic method signature.

```
class TAX<α1>extends Object {
  (int, α1) income;
  (int, α4) tax((int, α5) salary){self.income := salary; result := self.income*0.20; }}
```

Library class *TAX*<α_1> can be instantiated in multiple ways in the analysis of another class, for instance, by instantiating α_1 with a ground level, say H. The intention is that for any ground instantiation of $\alpha_1, \ldots, \alpha_5$ that satisfies the constraints, the body of *TAX* should be typable in the security type system.

The inference of class *Inquiry* takes place in the context of the library containing *TAX*. Because *TAX* is parameterized, the type of field *employee* is assumed to be *TAX*<β_1> and the level of *employee* is β_2. We could have chosen the level of *employee* to be H, but by typing rules, this would prevent access to a public field, say *name*, of *employee*. Note that the level of *est* is completely specified. The inferred type of method *overpay* is $(\alpha_6, () \multimap\!\langle \alpha_7\rangle\!\!\to \alpha_8) \mid K$. Some of the constraints in K generated by our inference algorithm are $\alpha_6 \le \alpha_8$, $\beta_1 \le \alpha_8$ and $\beta_2 \le \alpha_8$.

Suppose *Inquiry* is annotated differently, so that the type of the *employee* field is (*TAX*<H>, β_2), i.e., β_1 above has been instantiated to H. Then the level of the result of the call to *tax* (i.e., level α_8) will also be secret – this was predicted by the constraint $\beta_1 \le \alpha_8$. Suppose Q is an object of type *Inquiry*

[2] i.e., the level of *self*. This information is used to prevent leaks of the pointer to the current target object to other untrusted sources[3].

[3] This information, called the *heap effect*, is required to prevent leaks due to implicit flow via conditionals and method call [5, 3].

and suppose that its level is H. Then to prevent implicit information leakage due to the call to *overpay*, the return result should be H [3]. This can be seen from the constraint $\alpha_6 \leq \alpha_8$ with α_6, the level of Q, instantiated to H. Finally, if *employee* itself is H, then the constraint $\beta_2 \leq \alpha_8$ forces the level of the return result to be H as expected.

Section 3 formalizes the annotated language and discuss security typing rules for which noninterference can be shown. Next, section 4 considers a sublanguage of the annotated language with programs annotated with level variables; for this language we specify an algorithm that infers security types. We discuss restrictions of the language to handle undecidability of inference and formalize inference for inheritance and override of library methods in the current analysis unit in a way that maintains modularity. These restrictions and treatments will be illustrated by an example that overrides method *TAX.tax* in section 4.3.

3 Language and security typing

We use the sequential class-based language from our previous work [3]. The difference is in the annotated language, where classes and methods may be polymorphic in levels. This allows a library class to be used in more than one way. We make an explicit separation between a library class and a collection of additional classes that are based on the library.

First, some terms: a *unit* is a collection of class declarations. A *closed unit* is a collection of class declarations that is well formed as a complete program, that is, it is a class table. The library is a closed unit from which we need its polymorphic type signature, encoded in some auxiliary functions defined later. A program based on a library can consist of several classes which extend and use library classes and which may be mutually recursive. We use the term *analysis unit* for the classes to which the inference algorithm is applied. Due to mutual recursion, several classes may have to be considered together. An analysis unit must be well formed in the sense that the union of it with the library should form a closed unit.

The annotated language. We shall now define the syntax for library units and also adapt the security typing rules from our previous work to the present language. Essentially, a polymorphic library is typable if all of its ground instances are.

The grammar is in Table 1. Although identifiers with overlines indicate lists, some of the formal definitions assume singletons to avoid unilluminating complication.

Since the problem we want to address is secure information flow, all the programs are assumed to be well formed as ordinary code; i.e. when all levels are erased, including class parameters $<\overline{\lambda}>$. Typing rules for our Java-like language are standard and can be found in our previous paper [3]. It suffices to recall that a collection of class declarations, called a *class table*, is treated as a function CT so that $CT(C)$ is the code for class C. Moreover, *field*(f, C) gives the type of field f in class C and *mtype*(m, C) gives the parameter and return types for

Table 1. Language grammar

T	$::= \texttt{bool} \mid C$ (where C ranges over $ClassName$)
κ	$::= \texttt{H} \mid \texttt{L}$ (level constants)
λ	$::= \alpha \mid \kappa$ (level variable, constant)
U	$::= T{<}\overline{\lambda}{>}$ (we also use W and R for this category)
CL	$::= \texttt{class } C{<}\overline{\alpha}{>} \texttt{ extends } U\{(\overline{U}, \overline{\lambda})\ \overline{f};\ \overline{M}\}$
M	$::= U\ m\ (\overline{U}\ \overline{x})\{S\}$
S	$::= x := e \mid e.f := e \mid x := \texttt{new } U \mid x := e.m(\overline{e}) \mid S; S \mid U\ x := e \texttt{ in } S \mid$ $\texttt{if } e \texttt{ then } S \texttt{ else } S$
e	$::= x \mid \texttt{null} \mid \texttt{true} \mid e.f \mid e == e \mid e \texttt{ is } U \mid (U)e$

method m declared or inherited in C. Subtyping is invariant: $D \leq C$ implies $mtype(m, C) = mtype(m, D)$.

We use T to represent an ordinary data type, while U ranges over parameterized class types, which take the form $T{<}\overline{\lambda}{>}$. Declaration of a parameterized class binds some variables $\overline{\alpha}$ in the types of the superclass and fields:

$$\texttt{class } C{<}\overline{\alpha}{>} \texttt{ extends } D{<}\overline{\lambda}{>}\{(\overline{U}, \overline{\lambda'})\ \overline{f}; \overline{M}\}$$

All variables appearing in field declarations are bound at the class level. Thus the parameterized class declaration above must satisfy a *well formedness* condition: $\overline{\alpha} \supseteq var(\overline{\lambda}) \cup var(\overline{U}) \cup var(\overline{\lambda'})$. These variables, if appear in method declaration and body, are also bound in class level. The rest free variables are bound in method level for method polymorphism.

Field types, including security label, can be retrieved by a given function $lsfield(f, U)$ which returns the appropriate type of field f in a (possibly instantiated) class U. Thus for the declaration displayed above, $lsfield(f, C{<}\overline{\alpha}{>}) = (\overline{U}, \overline{\lambda'})$, and for any $\overline{\lambda''}$ of the right length, $lsfield(f, C{<}\overline{\lambda''}{>}) = (\overline{U}, \overline{\lambda'})[\overline{\alpha} \leftarrow \overline{\lambda''}]$. We require that $lsfield(f, U)$ is defined iff $field(f, T)$ is defined, and moreover[4] if $U \preceq U'$ and $lsfield(f, U')$ is defined, then $lsfield(f, U) = lsfield(f, U')$.

Method types, possibly polymorphic, need more delicate treatment. Types for methods are given using a signature function $lsmtype$ so that $lsmtype(m, U)$, for method m declared or inherited in class U returns m's signature and a set K of constraints in the form of inequalities between constants and variables. We require that $lsmtype(m, T{<}\overline{\lambda}{>})$ is defined iff $mtype(m, T)$ is, and in that case the signature takes the form $\lambda_0, (\overline{U}, \overline{\lambda_1}) \dashv\langle\lambda_2\rangle\!\!\rightarrow (R, \lambda_3) | K$. The signature expresses the following policy: If the information in "self" is at most λ_0 and the information in parameters is at most $\overline{\lambda_1}$ with type $\overline{U}$, then any fields written are at level λ_2 or higher and the result level is at least λ_3, with result type R provided that the constraints K are satisfied.

Following our previous work [3], we also require *invariance under subclassing*: if $U \preceq R$ and m is declared in R, then $lsmtype(m, U) = lsmtype(m, R)$ (regardless of whether m is inherited or overridden in U). Analogous to the sub-

[4] The security subtyping relation $\preceq$ is defined in Table 2.

Table 2. Auxiliary definitions

For T, T' such that $T \leq T'$, we define:

$$\begin{array}{lcl}
\mathit{instance}(T{<}\overline{\kappa}{>}, T) & = & T{<}\overline{\kappa}{>} \\
\mathit{instance}(T{<}\overline{\kappa}{>}, T') & = & \mathsf{let}\ \mathtt{class}\ T{<}\overline{\alpha}{>}\ \mathtt{extends}\ T''{<}\overline{\lambda}{>} = CT(T) \\
 & & \quad \mathsf{in}\ \mathit{instance}(T''{<}\overline{\lambda}[\overline{\alpha} \leftarrow \overline{\kappa}]{>}, T') \\
T{<}\overline{\kappa}{>} \preceq T'{<}\overline{\kappa'}{>} & \Leftrightarrow & \mathit{instance}(T{<}\overline{\kappa}{>}, T') = T'{<}\overline{\kappa'}{>} \\
\mathit{tcomp}(T{<}\overline{\lambda}{>}, T'{<}\overline{\lambda'}{>}) = & & \mathsf{let}\ \ T'{<}\overline{\lambda''}{>} = \mathit{instance}(T{<}\overline{\lambda}{>}, T') \\
 & & \quad \mathsf{in}\ \cup_i \{\lambda''_i \leq \lambda'_i ; \lambda'_i \leq \lambda''_i\}
\end{array}$$

classing requirement on methods in object-oriented languages, this is to ensure information flow security in the context of dynamic method dispatch.

The subtyping relation $\preceq$ must take polymorphism and information flow into account. For built-in type, we define `bool` $\preceq$ `bool`. Class subtyping can be checked using the $\preceq$ function defined in Table 2, which propagates instantiation of a class up through the class hierarchy. The definition uses another function, *instance*, that carries out this propagation and constructs a suitable instantiation of a supertype. The auxiliary definition for downcast appears in the appendix. For use in inference, we need to generate a set of constraints such that two types with variables are in the $\preceq$ relation; this is the purpose of function *tcomp*. We assume that if the analysis unit mentions a parameterized class, it provides the right number of parameters.

Security typing rules. Although the typing rules work with parameterized class declarations and with polymorphic method signatures, the typing rules for expressions and commands in method bodies only apply to ground judgements. A security type context Δ is a mapping from variable names to security types. We adopt the notation style for typing judgements from [3]. A judgement $\Delta \vdash e : (U, \kappa)$ says that expression e in context Δ has security type (U, κ). A judgement $\Delta \vdash S :$ `com` κ_1, κ_2 says that, in the context Δ, command S writes no variables below κ_1, which is in the store and will be gone after the execution of the method, and no fields below κ_2 , which will stay in the heap until garbage-collected.

We give the rule for method call, $x := e.m(\overline{e})$, below. It uses the polymorphic signature function of the method m, and requires that there must be some satisfying ground instance compatible with the levels at the call site; this is ensured by requiring satisfiability of a constraint set K' which contains the constraints needed to match security types of parameters and arguments.

$$\frac{\begin{array}{c}
\Delta, x : (U, \kappa) \vdash e : (W, \kappa_3) \\
\Delta, x : (U, \kappa) \vdash \overline{e} : (\overline{U}, \overline{\kappa_4}) \qquad \mathit{lsmtype}(m, W) = \lambda_0, (\overline{U'}, \overline{\lambda}) {-}\langle\lambda_1\rangle{\rightarrow} (U', \lambda_2) | K \\
\kappa_5 \leq \kappa \qquad \kappa_3 \leq \kappa \qquad K' = K \cup K'' \cup \mathit{tcomp}(U', U) \cup \mathit{tcomp}(\overline{U'}, \overline{U}) \\
K'' = \{\overline{\kappa_4} \leq \overline{\lambda}, \lambda_2 \leq \kappa, \kappa_6 \leq \lambda_1, \kappa_3 \leq \lambda_0, \kappa_3 \leq \lambda_1\} \qquad K' \text{ is satisfiable}
\end{array}}{\Delta, x : (U, \kappa) \vdash x := e.m(\overline{e}) : \mathtt{com}\ \kappa_5, \kappa_6}$$

Table 3. Typing rules for method declarations and class declarations

$$\frac{\begin{array}{c} \mathit{lsmtype}(m, C\texttt{<}\overline{\kappa'}\texttt{>}) = \lambda_0, (\overline{U}, \overline{\lambda}) \multimap\!\langle \lambda_3 \rangle\!\rightarrowtail (U, \lambda_4) | K \\ V = \mathit{vars}(\lambda_0, (\overline{U}, \overline{\lambda}) \multimap\!\langle \lambda_3 \rangle\!\rightarrowtail (U, \lambda_4) | K) \\ \text{for all } I \text{ with } \mathit{ok}(K, V, I) : \mathsf{let}\ \kappa_0, (\overline{U'}, \overline{\kappa}) \multimap\!\langle \kappa_3 \rangle\!\rightarrowtail (U', \kappa_4) = I(\lambda_0, (\overline{U}, \overline{\lambda}) \multimap\!\langle \lambda_3 \rangle\!\rightarrowtail (U, \lambda_4))\ \mathsf{in} \\ \overline{x} : (\overline{U'}, \overline{\kappa}), \texttt{self} : (C, \kappa_0),\ \texttt{result} : (U', \kappa_4) \vdash S : \texttt{com}\ \kappa_1, \kappa_2 \qquad \kappa_3 \leq \kappa_2 \end{array}}{C\texttt{<}\overline{\kappa'}\texttt{>}\ \texttt{extends}\ R \vdash U\ m(\overline{U}\ \overline{x})\{S\}}$$

$$\frac{\text{for all } M \in \overline{M}, \text{ all } \kappa\text{:} \quad C\texttt{<}\overline{\kappa}\texttt{>}\ \ \texttt{extends}\ I(R) \vdash I(M) \qquad \text{where } I = [\overline{\alpha} \leftarrow \overline{\kappa}]}{\vdash \texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \ \texttt{extends}\ R\{\overline{U}\ \overline{f}; \overline{M}\}}$$

Table 3 gives the rules for class and method declarations. A class declaration is typable provided that all of its method declarations are. Typing a method declaration requires checking its body with respect to all ground instantiations, I, over the variables V given by *lsmtype*. We define $ok(K, V, I)$ to mean that I satisfies K.

Noninterference. Like FlowCaml [15], our system uses a level-polymorphic language, both for more expressive libraries and because it is the natural result from inference. The noninterference property asserted by a polymorphic type is taken to be ordinary noninterference for all ground instances that satisfying the constraints that are part of the type. By lack of space in this paper, we omit the semantics and thus cannot formally define noninterference. Informally, a command is noninterfering if, for any two initial states that are indistinguishable for L (i.e., if all H fields and variables are removed), if both computations terminate then the resulting states are indistinguishable. Indistinguishability is defined in terms of a ground labeling of fields and variables. A method declaration is noninterfering with respect to a given type if its body is noninterfering, where the method type determines levels for parameters and result. A class table is noninterfering if, for every ground instantiation of every class, every method declaration is noninterfering.

The forthcoming technical report [18] shows that if a class table is typable by the security rules then it is noninterfering.

4 Inference

In this section we give the complete inference process. The algorithm has two steps. In the first step (sections 4.1 and 4.2) it outputs the constraints that ensure the typability of the classes being checked. In the second step (section 4.3), it takes the output from the first step, produces the parameterized signatures, and checks the subclassing invariance of these signatures. Then the new parameterized signatures can be added to the library.

4.1 Input

One input to the inference algorithm is the pair of auxiliary functions giving the polymorphic signatures of a library, namely, *lsfield* for fields and *lsmtype* for methods.

The other input is the current analysis unit. Unlike library methods, all methods implemented in the analysis unit are treated monomorphically with respect to each other during the inference, even though they may override polymorphic methods in the library. In particular, although we do not have explicit syntax for mutual recursion,[5] mutually recursive classes are put in the same analysis unit and are treated monomorphically. Method bodies can of course instantiate library methods differently at different call sites.

For any set V of variables, we write I^{*V} for some fixed renaming that maps V to distinct variables not in V.

The signature functions, *usfield* and *usmtype*, provide the types of fields and methods for classes in *unit*. We refrain from defining the simpler one, *usfield*. For any set V of level variables, define $\mathit{usmtype}(m, T, V)$ as follows:

1. If T has a declaration of m
   – If T has a superclass U in *unit* that declares m, then $\mathit{usmtype}(m, T, V) = \mathit{usmtype}(m, U, V)$.
   – Otherwise (i.e, any superclass of T that declares m is in the library), $\mathit{usmtype}(m, T, V)$ has parameter types and return type as declared in T; the heap effect and self level are two variables distinct from all level variables in *unit* and the signature has the empty constraint set.
2. If T inherits m from its superclass U
   – If U is in *unit*, $\mathit{usmtype}(m, T, V) = \mathit{usmtype}(m, U, V)$.
   – If U is a library class, $\mathit{usmtype}(m, T, V) = I^{*V} \mathit{lsmtype}(m, U)$.

By definition, *usmtype* may return a type that is either monomorphic or polymorphic, depending on whether there are any declarations of m in *unit* at or above T. If there are none, the method type is polymorphic and a renaming is needed to ensure variable freshness. On the other hand, if there is a declaration of m in *unit* at or above T, *usmtype* returns a fixed monomorphic type for all call sites – even if m has also been defined in the library.

4.2 Inference rules

The inference algorithm is presented in the form of rules for a judgment that generates constraints and keeps track of variables in use in order to ensure freshness where needed. For expressions, the judgment has the form

$$\Delta, V \vdash e : U \rightsquigarrow \alpha, K, V'$$

where V, V' are sets of level variables, with $V \subseteq V'$. The judgment means that in security type context Δ, expression e has type U and level α provided the

[5] In contrast with explicit syntax for mutual recursion, say, in ML

Table 4. Inference algorithm: selected command cases.

$$\frac{\begin{array}{c}\Delta, x : (U, \lambda_1) \vdash e : U' \rightsquigarrow \alpha_2, K_1, V_1 \\ K' = K_1 \cup \{\alpha_2 \leq \lambda_1, \alpha_3 \leq \lambda_1\} \cup \mathit{tcomp}(U', U) \qquad \alpha_3, \alpha_4 \notin V_1\end{array}}{\Delta, x : (U, \lambda_1), V \vdash x := e \rightsquigarrow \mathtt{com}\ (\alpha_3, \alpha_4), K', \{\alpha_3, \alpha_4\} \cup V_1}$$

$$\frac{\begin{array}{c}(\qquad (\lambda_0, (\overline{U'}, \overline{\lambda}) \!-\!\langle\lambda_1\rangle\!\!\rightarrow R, \lambda_2) | K = \mathit{usmtype}(m, W, V_1) \\ \vee \quad (\lambda_0, (\overline{U'}, \overline{\lambda}) \!-\!\langle\lambda_1\rangle\!\!\rightarrow R, \lambda_2) | K = I^{*V_1}(\mathit{lsmtype}(m, W)) \qquad) \\ \Delta, x : (U, \lambda), V \vdash e : W \rightsquigarrow \alpha_3, K_0, V_0 \qquad \Delta, x : (U, \lambda), V_0 \vdash \overline{e} : \overline{U} \rightsquigarrow \overline{\alpha_4}, K_1, V_1 \\ V'' = V_1 \cup \mathit{var}(K) \cup \mathit{var}(R) \cup \mathit{var}(\overline{U'}) \cup \mathit{var}(\lambda_0, \overline{\lambda}, \lambda_1, \lambda_2) \\ K' = K \cup K_0 \cup K_1 \cup \mathit{tcomp}(\overline{U}, \overline{U'}) \cup \mathit{tcomp}(R, U) \cup K'' \\ K'' = \{\overline{\alpha_4} \leq \overline{\lambda}, \alpha_3 \leq \lambda_0, \alpha_3 \leq \lambda_1, \lambda_2 \leq \lambda, \alpha_5 \leq \lambda, \alpha_6 \leq \lambda_1, \alpha_3 \leq \lambda\} \\ \alpha_5, \alpha_6 \notin V'' \qquad V' = \{\alpha_5, \alpha_6\} \cup V''\end{array}}{\Delta, x : (U, \lambda), V \vdash x := e.m(\overline{e}) \rightsquigarrow \mathtt{com}\ (\alpha_5, \alpha_6), K', V'}$$

constraints in K are satisfied. Each rule also has a condition to ensure freshness of new variables, e.g., $\alpha \notin V$. The constraints K may be expressed using other new variables; V' collects all the new and existing variables. The correctness property is that any ground instantiation I of V' that satisfies K results in an expression typable in the security type system, once we instantiate α and the other variables. This is formalized in the soundness theorem.

There is a similar judgment for commands: $\Delta, V \vdash S \rightsquigarrow \mathtt{com}\ (\alpha_1, \alpha_2), K', V'$ where V, V' are level variables with $V \subseteq V'$ means that in security type context Δ, command S writes to variables of level α_1 or higher and to fields of level α_2 or higher.

We refrain from giving the full set of rules, but discussing just a few cases, which are given in Table 4. The first rule in the table is for variable assignment. This may help the reader become familiar with the notation. The inferred type for the assignment is $\mathtt{com}\ (\alpha_3, \alpha_4)$, where α_3, α_4 are fresh. The generated constraint set K' contains the set K_1 obtained during the inference of e; U' is the type of e and α_2 is the inferred level of e. As expected from the typing rule for assignment, K' contains the constraint set $\{\alpha_2 \leq \lambda_1, \alpha_3 \leq \lambda_1\}$ and also the constraints between variables in U' and U generated by $\mathit{tcomp}(U', U)$, which ensures the subtyping relation between U' and U.

The most complicated rule is for method invocation (Table 4). It is developed from the typing rule for method invocation(Table 3). One can see that the conditions in the typing rule evolve to the constraints in the inference rule. There are two cases depending on the static type of the target. If the target is defined in the library, *lsmtype* will return the polymorphic method type and a renaming I^{*V} is used in the rule for freshness. Otherwise, *usmtype* returns the appropriate method signature, already renamed if necessary. In both cases, the type will be matched against the calling context and constraints in the returned signature will be integrated. The rule uses *tcomp* to generate constraints that ensure type compatibility.

Table 5. Inference algorithm for method declaration and class declaration

$$\frac{\Delta, V \vdash S \leadsto \texttt{com}\ (\alpha_1, \alpha_2), K_1, V_1 \qquad \Delta = [\overline{x} : (\overline{U}, \overline{\lambda}); \texttt{self} : (U, \lambda_0);\ \texttt{result} : (R, \lambda_4)] \quad \mathit{usmtype}(m, C, V_1) = (\lambda_0, (\overline{U}, \overline{\lambda}))\!-\!\langle\lambda_3\rangle\!\!\rightarrow\!(R, \lambda_4)|\varnothing}{C\ \texttt{extends}\ D, V \vdash R\ m(\overline{U}\ \overline{x})\{S\} \leadsto K_1 \cup \{\lambda_3 \leq \alpha_2\}, V_1}$$

$$\frac{\forall M_i \in \overline{M} \quad C\ \texttt{extends}\ D, V_{i-1} \vdash M_i \leadsto K_i, V_i}{V_0 \vdash \texttt{class}\ C\ \texttt{extends}\ D\{\overline{U}\ \overline{f}, \overline{M}\} \leadsto \cup_{1 \leq i \leq n} K_i, V_n}$$

The rules apply by structural recursion to a method body, generating constraints for its primitive commands (like assignment and method call) and constraints for combining these constituents (like in if/else). The rule for method declaration, first rule in Table 5, matches a method body with its declared type and checks it, generating an additional constraint.

The rule for class declaration, also in Table 5, combines the constraints for all its methods. We refrain from stating a formal rule for the complete analysis unit. The conclusion, written $\mathit{lsmtype}, \mathit{lsfield}, \mathit{usmtype}, \mathit{usfield} \vdash \mathit{unit} \leadsto K, V$ depends on two hypotheses. First, each class declaration in *unit* has been checked by the rule in the table, yielding constraints K over variables V. This check is obtained by enumerating the class declarations in *unit*, threading variable sets from one class to the next, and then taking for K the union of the constraints. The initial variable set contains all the fresh variables used in the definition of *usmtype* and all level variables that occur in *unit*. The second hypothesis is that overriding declarations do not introduce new constraints, which would invalidate the analysis of the library which is assumed in the form of *lsmtype*. If this check fails, the analysis fails. We will address the check at the end of section 4.3.

4.3 Building a new library

In this subsection we illustrate the manipulation of parameterized classes, resulting in a new library signature. Then we give the definitions. Finally, we outline how subclassing invariance is checked.

Producing new signatures. Assume we define a class *CreditTAX* that extends *TAX*. We have filled in level variables where needed.

```
class CreditTAX extends TAX<γ1> {
  (int, γ0) credit;
  (int, γ2)  tax((int, γ3) salary){
    self.income:=salary; result:=self.income*0.2-self.credit;}}
```

Assume $\mathit{usmtype}(\mathit{tax}, \mathit{CreditTAX}, \{\gamma_0, \gamma_1, \gamma_2, \gamma_3\})$ returns $(\gamma_s, (\mathit{int}, \gamma_3))\!-\!\langle\gamma_h\rangle\!\!\rightarrow\!(\mathit{int}, \gamma_2)|\varnothing$. We run the program on the code, and get the output K, V, where V includes $\gamma_0, \gamma_1, \gamma_2, \gamma_3$ and other temporary level variables generated during the inference.

To put *CreditTAX* into the library, we need to produce its signature. First we define the list of formal parameters by collecting variables γ_0 from field declarations and γ_1 from the "extends" clause. Second, we attach the generated constraint K to each method in the unit. The converted signature for *CreditTAX*, in pseudocode, is:

```
class $CreditTAX<\gamma_0, \gamma_1>$ extends $TAX <\gamma_1>$ {
  credit: $(\mathbf{int}, \gamma_0)$;
  tax:    $(\gamma_s, (\mathbf{int}, \gamma_2)) \multimap \langle\gamma_h\rangle \rightarrow (\mathbf{int}, \gamma_3) \mid K$ ; }
```

Now we formalize the process of producing new signatures. By the algorithm we can get (K, V) on the classes in the unit. For converting the code, let X be the set of class names declared in *unit*. We study any class $C \in X$. Let V' be all the variables in V that appear in the supertype or field type/label of C. Let *unit*$'$ be *unit* but with every C in X replaced by $C{<}V'{>}$. The *unit*$'$ is now a parameterized class with polymorphic methods.

Now we need to combine the signatures from the library and the unit. Based on *unit*$'$, we will build a new signature function that can access the converted unit and the library uniformly. Assume *unit*$'(T) =$ `class` $T{<}\overline{\alpha}{>}\{\ldots\}$.

```
$fieldmerge\ (lsfield, usfield)(f, T{<}\overline{\lambda}{>}) =$
    if $T$ in $unit$ then  $usfield(f, T)[\overline{\alpha} \leftarrow \overline{\lambda}]$
    else  $lsfield(f, T{<}\overline{\lambda}{>})$
$methmerge\ (lsmtype, usmtype, K)(m, T{<}\overline{\lambda}{>}) =$
    if $m$ is inherited from class $D$ in the library then
            $lsmtype(m, instance(T{<}\overline{\lambda}{>}, D))$
    else  $(fst(usmtype(m, T, \varnothing))|K)[\overline{\alpha} \leftarrow \overline{\lambda}]$
```

In *methmerge*, $T.m$ is implemented in the unit. So the third parameter for *usmtype* is insignificant and the constraint in the return of $usmtype(m, T, \varnothing)$ is empty. We use *fst* to strip off this empty constraint.

Checking method declarations for proper override. Rather than delving into algorithmic optimizations, we just specify the check for overriding declarations informally. We want to ensure that $U.m$ properly overrides $U'.m$ where U' is a super class of U. We assume the constraint set has been simplified in that only level constants and variables that are in the formal class parameter list or method type signature are kept. For example, $\{\alpha \leq \beta, \beta \leq \gamma\}$ can be transformed into $\{\alpha \leq \gamma\}$ if β is insignificant.

The condition for proper override can be expressed as: *Every constraint in the overriding method must be entailed [13] by the constraints in the overridden method.* For example, assume $lsmtype(U, m) = (L, () \multimap \langle\alpha_2\rangle \rightarrow \alpha_3)|K$ and $lsmtype(U', m) = (\beta_1, () \multimap \langle\beta_2\rangle \rightarrow \beta_3)|K'$. We want to check if $U.m$ is properly implemented. Since the level of *self* is L for $U.m$, $\beta_1 = L$ should be entailed by K'. Also, if $\alpha_2 \leq \alpha_3 \in K$, K' should entail it too.

We return to the *CreditTAX* example. It is not difficult to figure out that K is the same as the constraint set (after the name conversion) in *TAX.tax* except that there is one more inequality, $\gamma_0 \leq \gamma_3$, in K. This is necessary to ensure the typability of *CreditTAX*, but it makes the method *tax* more restrictive than declared in *TAX*. When *tax* is invoked on a *CreditTAX* object as an instance of *TAX*, the caller may assume $\gamma_0 = H, \gamma_3 = L$ as a valid precondition because *TAX.tax* does not impose any constraint between γ_0 and γ_3. But this constraint is obviously unsatisfiable for *CreditTAX.tax* in the context of dynamic dispatch, and violates the underlying policy. To make *CreditTAX* pass the check $\gamma_0 \leq \gamma_3$, one can relabel field *credit* with L.

We only compare constraints for a particular method — it is certainly not the case that the constraints from the library imply all constraints for *unit*, e.g., the unit can have additional methods.

Complexity The time/space cost for the inference algorithm to generate constraints is low-order polynomial in the size of the program, and independent of the security lattice. We can show that the time to generate the constraint set is $O(mn(s+t)^3 2^{2|P|})$, where m is the number of methods in the unit; n is the length of the unit; s, t are the number of distinct variable in class level and method level, perspectively; $|P|$ is the size of the permission set. The size of the generated constraint set is $O(n(s+t)^3 2^{2|P|})$.

5 Soundness and completeness of the inference algorithm

5.1 Soundness

Theorem 1 (Soundness of inference algorithm).
Assume $sigs \vdash unit \rightsquigarrow K, V$. [6] Let $unit'$ be the converted *unit* and $sfield = fieldmerge\ (lsfield, usfield)$ and $smtype = methmerge(lsmtype, usmtype, K)$ be the converted signatures, then $sfield, smtype \vdash unit'$.

5.2 Completeness of inference algorithm

In our system, the most general signatures of mutually recursive classes cannot be represented in finite forms. Thus the inference algorithm cannot be complete, since our algorithm will always terminate and produce finite output. We have to restrict the classes in current analysis unit in order to prove completeness.

Define a unit to be *monomorphically typed* if all type references and method invocations for the same class or method in a class body are instantiated exactly in the same way.

Theorem 2 (Completeness).
If $I(unit)$ is monomorphically typed in $I(sigs)$, the constraints produced by the algorithm for *unit* are satisfiable by an extension of I.

[6] We use *sigs* to abbreviate *lsfield*, *lsmtype*, *usfield*, *usmtype*.

In other words, this means that the algorithm yields principal types for a monomorphically typed unit with respect to the polymorphic library. This is analogous to type inference of recursive functions in ML. For example, in the ML term, $\mathsf{letrec}\ \ f(x) = t_1\ \mathsf{in}\ t_2$, all occurrences of f in t_1 are monomorphic. The current unit is comparable to t_1, and t_2 is comparable to classes in other units that can use current unit polymorphically once it has been made part of a library. The theorem relies on lemmas for expressions, commands, method and class declarations. We only list the lemma for expressions and commands.

Lemma 1. Assume $I(e)$ is monomorphically typed in $I(sigs)$. If $I(sigs), I(\Delta) \vdash I(e) : U_c, \kappa$ and $sigs, \Delta, V \vdash e : U \leadsto \alpha, K, V'$ where U_c is a type parameterized over level constants, then $\exists I' \supseteq I\ .\ ok(I', K, V') \wedge \kappa = I'(\alpha) \wedge U_c = I'(U)$

Lemma 2. Assume $I(S)$ is monomorphically typed in $I(sigs)$. If $I(sigs), I(\Delta) \vdash I(S) : \mathtt{com}\ \kappa_1, \kappa_2$ and $sig, \Delta, V \vdash S \leadsto (\mathtt{com}\ \alpha_1, \alpha_2), K', V'$, then
$\exists I' \supseteq I\ .\ ok(I', K', V') \wedge \kappa_1 = I'(\alpha_1) \wedge \kappa_2 = I'(\alpha_2)$.

6 Related Work and Discussion

Related Work. Volpano and Smith [19], give a security type system and a constraint-based inference algorithm for a simple procedural language. The type system guarantees noninterference: a well-typed program does not leak sensitive data. The inference algorithm is sound and complete with respect to the type system. However, they do not handle object-oriented features, and their suggestion to handle library polymorphism by duplicating code is impractical.

Myers [9, 10] gives a security type system for full Java, but leaves open the problem of justifying the rules with a noninterference result. Myers, Zdancewic and their students have implemented a secure compiler, Jif,[7] that implements the security typing rules. Jif handles several advanced features like constrained method signature, exceptions, declassification, dynamic labels and polymorphism. Jif's inheritance allows overriding methods to be more general than overridden methods, which means that the constraints in the overridden method must be stronger than the overriding method. However, inference in the system is only intraprocedural. Field and method types are added either manually or by default.

Simonet presents a version of ML with security flow labels, termed FlowCaml[16, 15] which supports polymorphism, exceptions, structural subtyping and the module system. The type system is polymorphic and has been shown to ensure noninterference. Simonet and Pottier[12] give an algorithm to infer security types. They also prove soundness of type inference.

There is a rich literature on type inference for object-oriented programs [20, 11, 2, 4, 21]. However, we are interested in security type inference, rather than full type inference; we assume that a well-typed program is given. We found it

[7] On the web at `http://www.cs.cornell.edu/jif/`

difficult to adapt the techniques in these works because they do not consider modular inference in the presence of libraries.

We have a working prototype for a whole program analysis for the language in [3]. It accepts a class declaration that is partly annotated with level constants, generating a constraint set and checking its satisfiability. If the code is typable, the output will be a polymorphic type for the given program in its most general form. The extension of the prototype for the present paper is currently under way.

Deployment model. For an application developer, the signatures in the library specify security requirements. The developer must annotate additional methods in the current analysis unit with new policies. Running a check on the annotated program can then tell whether it is secure with respect to the library policies.

For library designers, the tool is helpful in that it not only enforces the specified security policies, but also gives designers a chance to revise the result signatures if the signatures appear too *general* and seem likely to prevent subclasses from being implemented because subclasses cannot introduce new flows.

To make the result signatures more general for a collection of classes, it is advisable to make the analysis unit as small as possible. Classes that make mutually recursive references need to be analyzed together. This is the only reason to make units have more than one class.

Conclusion. The main contribution of this paper is the specification of a modular algorithm that infers security types for a sequential, class-based, object-oriented language. This requires the addition of security level variables to the language and moreover, requires classes parameterized with security levels. The inference algorithm constructs a library where each class is parameterized by the levels in its fields. Each method of a parameterized class can be given a polymorphic, constrained signature. This has the additional benefit of being more expressive and flexible for the programmer. We have given soundness and completeness theorems for the algorithm and work is in progress on a prototype. We have not yet experimented with the scalability of our technique to real sized programs. Such an experiment and its results will be reported in the first author's dissertation. Our work would also benefit from a comparison with the HM(X) constraint-based type inference framework [17]. Our suspicion, however, is that to prove soundness and completeness, there might be substantial overhead in the translation of our security types to the HM(X) framework.

References

1. Martin Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.
2. Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object Oriented Programming (ECOOP)*, pages 2–26, 1995.

3. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–270. IEEE Computer Society Press, 2002.
4. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.
5. Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
6. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
7. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.
8. Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Sixth International Symposium on Programming*, number 166 in Lecture Notes in Computer Science. Springer-Verlag, 1984.
9. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.
10. Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control.* PhD thesis, Laboratory of Computer Science, MIT, 1999.
11. Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1991.
12. François Pottier and Vincent Simonet. Information flow inference for ML. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 319–330, 2002.
13. Jakob Rehof and Fritz Henglein. The complexity of subtype entailment for simple types. In *Proceedings LICS '97, Twelfth Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland*, June 1997.
14. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
15. Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.
16. Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.
17. Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Proceedings of the Workshop on Types in Programming (TIP'02)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.
18. Qi Sun, Anindya Banerjee, and David A. Naumann. Constraint-based security flow inferencer for a Java-like language. Technical Report KSU CIS TR-2004-2, Kansas State University, 2004. In preparation.
19. Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, number 1214 in Lecture Notes in Computer Science, pages 607–621. Springer-Verlag, 1997.
20. Mitchell Wand. Complete type inference for simple objects. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.
21. Taejun Wang and Scott Smith. Precise constraint-based type inference for java. In *European Conference on Object Oriented Programming (ECOOP)*, 2001.