Information Flow Control with Errors

Andreas Gampe
The University of Texas at San Antonio
agampe@cs.utsa.edu

Jeffery von Ronne
The University of Texas at San Antonio
vonronne@cs.utsa.edu

Abstract

Information Flow Control is concerned with the correct handling of data with respect to a security policy. A common enforcement technique is annotated type systems. For object-oriented languages, type systems have been developed for class-based languages. The reason for this is that in a class-based language, it is simpler to design a type system that ensures well-typed programs do not result in method-not-found errors, and this property can be assumed in the information flow control mechanism.

In the case of dynamic languages, for example, prototype-based ones like Javascript, no type system is currently powerful enough to handle common language idioms, which hinders the adoption of security-typing in practical settings. As a solution this paper proposes to make the handling of method-not-found errors explicit in the security type system: the type system does not enforce regular soundness, so well-typed programs might fail, but even in case of such errors non-interference is ensured. This paper outlines this approach and provides an initial investigation of its feasibility. A security type system for a functional object calculus with extension is presented and shown to enforce non-interference.

1. Introduction

As society increasingly makes use of networked computer systems that store, process and transmit sensitive information, it has become more important to formally prove that programs handle data in an appropriate fashion. Many different security properties have been proposed in the past. A common property is noninterference. Informally, a program has the noninterference property, if private inputs do not influence public outputs. Then, under the assumption that an attacker can only access public outputs, she cannot deduce which inputs were used. There are two main approaches to enforce noninterference: one is static and the other is dynamic. Both control the ways information is allowed to flow in the program and are called static and dynamic information flow control, respectively.

In the dynamic approach, all information is labeled at runtime. Whenever information is used, labels on the results are set accordingly. Only low-labeled results are allowed to be sent to low outputs. Besides a difficulty with certain forms of hidden flows, there can be significant overhead in the label computations (cf. [5]).

In the static context, a type regime is established. Types are annotated with conservative approximations of the labels of runtime values. Type checking will enforce the correct handling of data. A big advantage is the low overhead at runtime. It is also possible to retain the annotation of source programs and use these for a lightweight verification of the programs on a user's computer.

Several languages have been proposed for static information flow control. Where proofs have been given, they are usually established in the standard manner of progress and subject reduction (or preservation). The underlying type system is either proven sound in the process, or assumed so: any program that can be typed in the underlying program cannot produce errors during execution. Thus it is unnecessary to include errors in the information flow control layer on top of the underlying type system.

In the object-oriented setting, this means there are no message-not-found errors at runtime. This is a strong guarantee, which is hard to prove for many practical, dynamically-typed languages. In fact, recent empirical studies [20] showed that in the case of Javascript, most of the simplifying assumptions made in state-of-the-art work are violated in practice. A general type system that does not reject a significant number of practical programs seems out of reach at the moment. How is it then possible to prove noninterference statically?

We propose to explicitly handle the case of method-not-found errors in the computation. In this model, it is acceptable if a program fails because a method is to be called that the target object does not possess. However, errors cannot be allowed to transport information about private inputs.

We adopt the first-order version $\mathrm{Ob}_{1<:}$ of the Object Calculus of Abadi and Cardelli [1]. This calculus is centered around objects as the only primitives. The allowed operations are object formation, method invocation, and method override. We extend the semantics to also allow method extension (i.e., adding new methods) and make errors explicit. Differing from previous work however, our target is not type soundness in the usual sense. Our types are overapproximations of the interface of an object. If an object can be typed, it will have a subset of the methods. Thus the meaning of an object type is that the invocation of a method either returns an object of the given method return type, or results in a method-not-found error. This allows us to control the invocation even of methods that do not exist, in a principled manner. If such a method is added through extension, it needs to agree with the previously stored type. Therefore, old judgments are still valid. To make the scheme work, we allow errors to be subtypes of all types.

This paper is structured as follows: Section 2 gives an intuition about our long-term goal, using simple examples. In section 3 we define the syntax and semantics of our calculus. Section 4 describes the type system we employ to enforce noninterference. A proof of noninterference for our calculus is given in section 5. Related work is discussed in Section 6. Section 7 concludes with remarks on ongoing work.

FOOL '11 June 5th 2011, San Jose.
Copyright © 2011 ACM [to be supplied]. . . $10.00

2. Example

In this section, we give a high-level example of what we are trying to accomplish. For simplicity, we assume a simple, imperative, object-based language that supports method extension. Then we could write a program like

```
o = ...
o.m()
```

With a standard type system, it might not be possible to establish that method m is defined when execution reaches the second line. Thus a standard type system approach will reject this program.

However, a missing method does not necessarily mean that there is a malicious flow of information, the usual suspect just being a bug. We propose to ignore the responsibilities of the underlying type system (i.e., ensuring the absence of method-not-found errors) and focus on information flow control even in the presence of such errors. There are two ways the program above might be safe: if we know that there is a method m, regardless of what high-level input the program received, or if we know there is no such method. In the first case, the computation will continue (possibly failing inside the call). In the second case, no matter the input, computation will fail at the second line. Thus, an attacker cannot gain more information than already available at that point.

Furthermore, if we cannot prove a method's existence or non-existence, but can show that that decision is only influenced by low conditions, a fail or successful call also does not leak information. Take for example the following program:

```
o = ...
if (b)
  o.n = function()...
o.n()
```

If the condition b is low, i.e., it is not confidential, an attacker will already know if the call will fail or not [1].

We propose a type system similar to a conservative analysis: types overapproximate the set of methods of a value. This allows us to type incomplete objects. In comparison to work in that area (see related work), we are more lenient with respect to method calls. For safety, previous work only allows a method call if it is statically known the method exists. We however do allow calls to all methods, since we do want to allow method-not-found errors. The reduction will terminate in an error state in the case of a missing method.

Different from the work on incomplete objects, we cannot allow an object's type to "forget" methods by subsumption. For our approximation to be safe, we can only increase the approximation.

In this work, we develop a type system in a simple first-order object calculus. In an imperative setting, information can be leaked through control flow structures and needs to be handled specially. In a functional setting, there are no control structures. Information gained through conditional execution (encoded through method calls) is transferred in the current object and never-decreasing. This eases the exposition of the idea, because if a call may fail in a high setting, the overall result of the program had to be high, too. For the imperative setting, further techniques (e.g., [3]) need to be integrated.

But encouragingly, our type system is already able to type incomplete objects and enforce noninterference in the presence of errors. Currently, ongoing work investigates a more expressive type system closer to our goal in an imperative calculus.

[1] If the call is made in a high environment, though, information would be leaked. A program position is in a high environment, if reaching this location depends on some confidential data. This is loosely related to [3].

$o ::=$	s	Variable
	$[m_i = \varsigma(s_i)e_i]^{\phi}_{i\in I}$	Object
	$o.^{\phi}m$	Method invocation
	$o \leftarrow^{\phi} m = \varsigma(s)o'$	Method override or extension
	err	Error constant

Figure 1. Extended Syntax

3. Base Calculus

As mentioned, we adopt the first order calculus of Abadi and Cardelli [1]. This calculus is functional and centered around primitive objects. The calculus consists essentially of four elements: variables, object literals, method invocations and method overrides.

Object literals are records with method names and method bodies. Bodies are functions that take the object itself as their single argument (self-application semantics). Method bodies are only processed when invoked. The syntax uses the ς binder instead of λ to signal this late-binding semantics.

Method invocation selects a method by name from an object, and substitutes the self-parameter with the object in the method body. Method override replaces the function stored in an object. In Abadi's calculus, method override is restricted to replacing an already existing method. We add method extension to allow the extension of an object with new methods, which is a common feature in practical dynamic object-based languages.

This simple calculus does not allow method parameters besides the self-parameter. However, multi-argument functions can be emulated through additional methods that stand for parameters. Method override of those methods will emulate parameter passing. The same technique can also be used to encode the simple lambda calculus in the object calculus. The calculus does not directly support delegation and prototypes. However, a form of delegation can also be encoded. Alternatively, the calculus can be extended, as for example done in [2]. For a more detailed account of the encodings we refer to [1].

For information flow control, we assume the simple security lattice [13] $(L, \sqsubseteq)$ with bottom element $\perp$ and join operation $\sqcup$. Elements of the lattice are used as annotated labels in the calculus and represent confidentiality levels. Labels are ranged over by ϕ and ψ. The partial order $\sqsubseteq$ then orders the labels with respect to the sensitivity of the information: If $\phi \sqsubseteq \psi$, then any value labeled ϕ is assumed less confidential than a value labeled ψ, and any value tagged with ϕ can be used in places where a value with label ψ is needed. As an example, take $L = \{l, h\}$ with the meaning that $l = \perp$ stands for low=public information, and h for high=private. Then $l \sqsubseteq h$, so that 5^l is less confidential than 4^h. For a more detailed description we refer to [13, 22].

We add security annotations to most of the syntactic elements. We include an explicit element for error. This element is not intended to be used by the programmer. The extended syntax is given in Figure 1.

We extend the purely reduction-based semantics of [1] to retain annotations. Furthermore, our calculus allows the addition of methods. The reduction rules $(Red - Over)$ and $(Red - Ext)$ are applied depending on the shape of the object being extended. In the inherited case, $(Red - Over)$ applies when the object already has a method of the given name. Otherwise, $(Red - Ext)$ will add the method to the object. Since the calculus is functional, a new object will be returned in either case. It is noteworthy that both cases will be handled by a single typing rule. There is no reduction rule for variables (they are discharged through substitution), and no rule that reduces an object literal. All other terms are erroneous and will reduce to err. The non-error reduction rules are listed in Figure 2. The error reduction rules are listed in Figure 3. Evaluation is

(With $o = [m_i = \varsigma(s_i)o_i]^{\phi}_{i \in I}$)

$(Red - Inv)$

$$o.^{\phi} m_j \quad \to \quad o_j\{s_j := o\} \qquad j \in I$$

$(Red - Over)$

$$o \leftarrow^{\psi} m_j = \varsigma(s_j)o' \quad \to \qquad j \in I$$
$$[m_i = \varsigma(s_i)o_i, m_j = \varsigma(s_j)o']^{\phi}_{i \in I \setminus \{j\}}$$

$(Red - Ext)$

$$o \leftarrow^{\psi} m_j = \varsigma(s_j)o' \quad \to \qquad j \notin I$$
$$[m_i = \varsigma(s_i)o_i, m_j = \varsigma(s_j)o']^{\phi}_{i \in I}$$

Figure 2. Reduction (without errors)

(With $o = [m_i = \varsigma(s_i)o_i]^{\phi}_{i \in I}$)

$$\begin{array}{llcll}
(Red - NotFound) & o.^{\phi} m_j & \to & \mathrm{err} & j \notin I \\
(Red - ErrInv) & \mathrm{err}.^{\phi} m_j & \to & \mathrm{err} & \\
(Red - ErrOver) & \mathrm{err} \leftarrow^{\psi} m_j = \varsigma(s_j)o' & \to & \mathrm{err} &
\end{array}$$

Figure 3. Reduction (errors)

derived from reduction by contracting the leftmost, outermost redex. This simplifies the exposition and corresponds to the regular evaluation strategy of [1].

4. Type System

The goal of our type system is to enforce noninterference, not the absence of runtime errors (e.g., method-not-found errors). Thus it works in an irregular fashion more reminiscent of a conservative analysis. Types are ranged over by τ and σ, and are labeled. An object type $[m_i : \tau_i]^{\phi}_{i \in I}$ describes objects with an annotation at most ϕ and a subset of the methods m_i. Thus, all possible callable methods are contained in the type. We made errors explicit with the error constant err. The type of errors is E. Since an object type is an over-approximation of an object's interface, we include a subtyping relationship of E with every type. Thus, any method may potentially return err [2]. This makes the approximation safe with respect to the reduction. The type E and the error element err are implicitly labeled with $\bot$ and not a valid type for components of an object type. We use the notation $\tau^{\sqcup \phi}$ for the type that results from τ by replacing the label with the join of the old label and ϕ, e.g., $(\mathtt{int}^{L})^{\sqcup H} = \mathtt{int}^{L \sqcup H} = \mathtt{int}^{H}$.

Type environments Γ store assignments of types to variables in the usual way. The error type is not allowed in a type environment.

We have the usual judgments:

$\Gamma \vdash *$	Well-formed environment
$\vdash \tau$	Well-formed type
$\vdash \tau_1 \leq \tau_2$	Subtypes
$\Gamma \vdash o : \tau$	Term has type

Well-formedness is defined in the usual way. Note that we do not need the type environment for type-related judgments, since we do not support type variables.

We add subtyping rules that correspond to our semantics of types. If an object is of a type with a set of methods, it is also of a type with the same list of methods, extended by a new method. This unusual subtyping is safe because of the object-has-subset

[2] In a sense, this is similar to null and the null-type in practical languages.

$(Sub - Refl)$
$$\frac{\vdash \tau}{\vdash \tau \leq \tau}$$

$(Sub - Trans)$
$$\frac{\vdash \tau \leq \sigma \quad \vdash \sigma \leq \tau'}{\vdash \tau \leq \tau'}$$

$(Sub - Err)$
$$\frac{\vdash \tau}{\vdash \mathrm{E} \leq \tau}$$

$(Sub - Partial)$
$$\frac{\vdash [m_i : \tau_i]^{\phi}_{i \in I} \quad \phi \sqsubseteq \psi}{\vdash [m_i : \tau_i]^{\phi}_{i \in I} \leq [m_i : \tau_i]^{\psi}_{i \in I}}$$

$(Sub - Ext)$
$$\frac{\vdash [m_i : \tau_i, m_j : \tau_j]^{\phi}_{i \in I, j \in J}}{\vdash [m_i : \tau_i]^{\phi}_{i \in I} \leq [m_i : \tau_i, m_j : \tau_j]^{\phi}_{i \in I, j \in J}}$$

Figure 4. Subtyping

$(T - Prof)$
$$\frac{\Gamma, s : \tau, \Gamma' \vdash *}{\Gamma, s : \tau, \Gamma' \vdash s : \tau}$$

$(T - Sub)$
$$\frac{\Gamma \vdash o : \sigma \quad \vdash \sigma \leq \tau}{\Gamma \vdash o : \tau}$$

$(T - Obj)$
$$\frac{\tau = [m_j : \sigma_j]^{\phi}_{j \in J} \quad I \subseteq J \quad \forall i \in I.\, \Gamma, s_i : \tau \vdash o_i : \sigma_i}{\Gamma \vdash [m_i = \varsigma(s_i)o_i]^{\phi}_{i \in I} : \tau}$$

$(T - Inv)$
$$\frac{\Gamma \vdash o : [m_i : \tau_i]^{\phi}_{i \in I} \quad j \in I}{\Gamma \vdash o.^{\psi} m_j : \tau_j^{\sqcup \psi \sqcup \phi}}$$

$(T - Over)$
$$\frac{\tau = [m_i : \tau_i]^{\phi}_{i \in I} \quad \Gamma \vdash o : \tau \quad \Gamma, s_j : \tau \vdash o_j : \tau_j \quad j \in I}{\Gamma \vdash o \leftarrow^{\psi} m_j = \varsigma(s_j)o_j : \tau^{\sqcup \psi}}$$

Figure 5. Typing Rules

semantics. Furthermore, we lift the partial order of the security lattice to subtyping. We finalize subtyping with the usual reflexivity and transitivity. The subtyping rules are shown in Figure 4.

The typing rules are an adapted version of the Object Calculus'. Projection types variables according to the type environment. A subsumption rule is defined in the standard way. The rule $(T{-}Obj)$ types an object literal. We can type an object with type τ, if τ contains at least all the methods in the body, and we can type all method bodies under the assumption that the self parameter is of type τ. The $(T - Inv)$ rule types an invocation if we can type the receiver with a type that contains the method. The $(T - Over)$ rule is used both for extension and override. In the type system, they are identical. To extend a method means to replace an ("abstract") method already contained in the type. The typing rules are listed in Figure 5.

5. Noninterference

We proceed to establish noninterference along the following lines: First, we show subject reduction for our type system. Next, we show that reduction and substitution are orthogonal. Then we show that low-typed terms in high environments are either values or can be reduced. Finally, we can establish a version of noninterference.

For this treatment, we need a helper function for labels of types and type environments. $\mathrm{toplabel}(\tau)$ is the outermost annotation of type τ. The judgment $\mathrm{toplabel}(\Gamma) \not\sqsubseteq \phi$ is defined as $\forall (s : \tau) \in$

$\Gamma.\mathrm{toplabel}(\tau) \not\sqsubseteq \phi$. Informally that means that all variables bound in the type environment are more confidential than the given label.

The proof of subject reduction structurally follows [1]. However, our semantics is small-step. We start with three standard auxiliary lemmas that are straightforward by induction and definition of the calculus.

LEMMA 5.1 (Generation). *If $\Gamma \vdash [m_i = \varsigma(s_i)o_i]^{\phi}_{i \in I} : \tau$, then there exists a type $\sigma = [m_j : \tau_j]^{\phi}_{j \in J}$ with $I \subseteq J$, such that $\forall j \in I.\ \Gamma, s_j : \sigma \vdash o_j : \tau_j$ and $\Gamma \vdash [m_i = \varsigma(s_i)o_i]^{\phi}_{i \in I} : \sigma$ and $\vdash \sigma \leq \tau$.*

LEMMA 5.2 (Weakening). *If $\Gamma, s : \tau, \Gamma' \vdash o : \sigma$ and $\vdash \tau' \leq \tau$ are derivable, then also $\Gamma, s : \tau', \Gamma' \vdash o : \sigma$.*

LEMMA 5.3 (Substitution). *If $\Gamma, s : \tau, \Gamma' \vdash o : \sigma$ and $\Gamma \vdash p : \tau$ are derivable, then also $\Gamma, \Gamma' \vdash o\{s := p\} : \sigma$.*

We are now able to prove the subject reduction property.

THEOREM 5.1 (Subject Reduction). *If $\Gamma \vdash o : \tau$ and $o \to o'$, then $\Gamma \vdash o' : \tau$.*

Proof By induction on the derivation of $\Gamma \vdash o : \tau$.

Case $(T - Prof)$
In that case, o is a variable and there is no reduction.

Case $(T - Sub)$
The inductive hypothesis applies to the first premise. Another application of $(T - Sub)$ yields the result.

Case $(T - Obj)$
Object literals are values and cannot be reduced.

Case $(T - Inv)$
Then $o = p.^{\phi}m_j$. There are three sub-cases.

Error
If p is an error or a constant, then the term is reduced to err. The result is obtained by application of $(T - Sub)$.

Not a value
If p is not a value, the inductive hypothesis applies to the first premise and $p \to p'$. Thus, $o' = p'.^{\phi}m_j$ can be typed with τ.

Value
In case p is a value, i.e., an object literal, there are two cases. If j is not a valid index, the term is reduced to err and subtyping will give the right result. If j is a valid index, then by the first premise we have a typing of the object. This allows us to apply the generation lemma, which yields $\Gamma \vdash p : \sigma$ and $\Gamma, s_j : \sigma \vdash o_j : \tau_j$. From the substitution lemma it follows that $\Gamma \vdash o_j\{s_j := p\} : \tau_j$. The result follows by subtyping.

Case $(T - Over)$
Then $o = p \leftarrow^{\phi} m_j = \varsigma(s_j)o_j$. Again, we have the three cases for p, where the error/constant and non-value cases are analogous. In the case that p is a value, we have a typing for the literal by the first premise. Applying the generation lemma gives us typings of the already established methods with a type $\sigma \leq \tau$. We can use the weakening lemma with the second premise of $(T - Over)$ and receive a typing of the new method with respect to σ. We can combine all those results to type the object literal that is the result of either $(Red - Over)$ or $(Red - Ext)$, according to whether a method was already there. Subtyping completes the case.

We now show that reduction and substitution are orthogonal.

LEMMA 5.4 (Subst Eval). *If $\Gamma, s : \sigma \vdash o : \tau$, $o \to o'$, and $\Gamma \vdash p : \sigma$, then $o\{s := p\} \to o'\{s := p\}$.*

Proof By induction on the derivation of $\Gamma, s : \sigma \vdash o : \tau$. Most cases are either vacuous or straightforward. We show the case of $(T - Inv)$, where $o = q.^{\phi}m_k$ and q is an object literal with method m_k. Then $o' = o_k\{s_k := q\}$ and $o\{s := q\} = (q\{s := q\}.^{\phi}m_k$. The substitution does not change the set of methods, so $(Red - Inv)$ can be applied. We need to distinguish the cases $s = s_k$ and $s \neq s_k$ when we apply the reduction. Finally, rewriting the term according to the standard substitution rules results in $o_k\{s_k := q\}\{s := p\}$.

The next to last step is progress of lowly-typed terms in high environments.

LEMMA 5.5 (Low Progress). *If $\Gamma \vdash o : \tau$ with $\mathrm{toplabel}(\Gamma) \not\sqsubseteq \mathrm{toplabel}(\tau)$, then either o is a value (object literal, constant or err), or there is an o' so that $o \to o'$.*

Proof By induction on the derivation of $\Gamma \vdash o : \tau$.

Case $(T - Prof)$
In that case, o is a variable and $(o = s : \tau)$ is part of the type environment. Thus, $\neg(\mathrm{toplabel}(\Gamma) \not\sqsubseteq \mathrm{toplabel}(\tau))$, so the second premise of the lemma is not fulfilled.

Case $(T - Sub)$
Subtyping maintains the partial order of security labels. Thus the inductive hypothesis applies to the first premise. So there exists an o' such that $o \to o'$.

Case $(T - Obj)$
Object literals are values, which fulfills the lemma.

Case $(T - Inv)$
Then $o = p.^{\phi}m_j$. There are three sub-cases.

Error
If p is an error or a constant, then the reduction $p.^{\phi}m_j \to$ err applies.

Not a value
If p is not a value, the inductive hypothesis applies to the first premise (same type, same environment) and $p \to p'$. Thus, $p.^{\phi}m_j \to p'.^{\phi}m_j$.

Value
In case p is a value, i.e., an object literal, there are two cases. If j is not a valid index, the reduction that applies is $o \to$ err. If j is a valid index, then we can apply $(Red - Inv)$. Thus we get $o \to o_j\{s_j := p\}$.

Case $(T - Over)$
Then $o = p \leftarrow^{\phi} m_j = \varsigma(s_j)o_j$. Again, we have the three cases for p, where the error and non-value cases are analogous. In the case that p is a value, we can either apply $(Red - Over)$ or $(Red - Ext)$, depending on the existence of m_j in p.

Finally, we approach noninterference. In the original calculus there are only primitive objects. To state noninterference, we need to define an equivalence relation on objects with respect to a given labeling. A simple definition is observational equivalence, where an attacker is allowed to invoke only low-typed methods, which corresponds to the ability of an attacker to inspect low-typed heap elements in an imperative model. The definition is slightly complicated by the fact that methods in literals are not annotated with types.

Let $\Downarrow$ denote the "big-step" closure of the reduction, i.e., $o \Downarrow v$ iff v is a value and $o \to^{*} v$. Note that the reduction rules are deterministic, so there is at most one value v. Furthermore, let $o \Downarrow \infty$ stand for an infinite, non-terminating computation, i.e., there is no value v such that $o \to^{*} v$.

DEFINITION 5.1 (Object Equivalence). *Two objects o_1 and o_2 are equivalent with respect to security level ϕ, written as $o_1 \sim_{\phi} o_2$, iff* *for all types $\tau = [m_i : \tau_i]^{\psi}_{i \in I}$ with $\psi \sqsubseteq \phi$, $\emptyset \vdash o_1 : \tau$ and $\emptyset \vdash o_2 : \tau$: For all methods m_i with* $\text{toplabel}(\tau_i) \sqsubseteq \phi$*:*

1. *If $o_1.^{\perp}m_i \Downarrow v_1$, then for some v_2, $o_2.^{\perp}m_i \Downarrow v_2$ and $v_1 \sim_\phi v_2$.*
2. *If $o_2.^{\perp}m_i \Downarrow v_2$, then for some v_1, $o_1.^{\perp}m_i \Downarrow v_1$ and $v_1 \sim_\phi v_2$.*

Now we can state our non-interference theorem.

THEOREM 5.2 (Noninterference). *If $s : \sigma \vdash o : \tau$ (where $\text{toplabel}(\tau) = \phi$) with $\text{toplabel}(\sigma) \not\sqsubseteq \phi$, $\emptyset \vdash p : \sigma$ and $\emptyset \vdash p' : \sigma$, then either both computations $o\{s := p\}$ and $o\{s := p'\}$ diverge, or both computations return results equivalent with respect to ϕ.*

Proof If $\text{toplabel}(\sigma) \not\sqsubseteq \text{toplabel}(\tau)$, we can apply the low-progress lemma to o. With subject reduction, we can repeat the process. This results in a chain of reductions $o \to o' \to o'' \to \ldots$, that is either finite and ends with a value (by the low-progress lemma), or that is infinite, i.e., the computation diverges.

We can now apply the subst-eval lemma to this chain for both p and p'. The result is, that the computations for $o\{s := p\}$ and $o\{s := p'\}$ diverge exactly if the computation of o diverges.

Now we can handle the case of converging computations. By subst-eval, the computations are $o \Downarrow v$, $o\{s := p\} \Downarrow v\{s := p\}$ and $o\{s := p'\} \Downarrow v\{s := p'\}$ for some v. To show equivalence, we use bisimulations, or more precisely the technique of strong bisimulations up to $\sim$ [16]. We choose the following binary relation:

$$S_\phi = \left\{ \begin{array}{c|r} (q\{s := p\}, & \exists \tau, \sigma.\ s : \sigma \vdash q : \tau \wedge \\ \quad q\{s := p'\}) & \vdash p : \sigma \wedge \vdash p' : \sigma \\ & \text{toplabel}(\sigma) \not\sqsubseteq \phi \wedge \\ & \text{toplabel}(\sigma) \not\sqsubseteq \text{toplabel}(\tau) \end{array} \right\}$$

We need to show that S_ϕ is closed under invocation of low-typed methods, that is, if $(q\{s := p\}, q\{s := p'\}) \in S_\phi$, and $s : \sigma \vdash q : [m_i : \tau_i]^{\psi}_{i \in I}$, then for all m_i with $\text{toplabel}(\tau_i) \sqsubseteq \phi$ with $q\{s := p\} \Downarrow v$ and $q\{s := p'\} \Downarrow v'$ we have $(v, v') \in S_\phi$.

The approach is similar to the one above. First, since $\text{toplabel}(\tau_i) \sqsubseteq \phi$, $\text{toplabel}(\sigma) \not\sqsubseteq \text{toplabel}(\tau_i)$. Thus we can apply low-progress, subject reduction and subst-eval. As above, either both computations return results or diverge. If they converge, the computations are $q.^{\perp}m_i \Downarrow v_0$, $q\{s := p\}.^{\perp}m_i \Downarrow v_0\{s := p\}$ and $q\{s := p'\}.^{\perp}m_i \Downarrow v_0\{s := p'\}$ for some v_0. Now $s : \sigma \vdash v_0 : \tau_i$, $\vdash p : \sigma$, $\vdash p' : \sigma$, $\text{toplabel}(\sigma) \not\sqsubseteq \phi$ and $\text{toplabel}(\sigma) \not\sqsubseteq \text{toplabel}(\tau_i)$. Thus, $(v, v') \in S_\phi$. This concludes the proof.

Note that err is different from all other elements. Thus, if one computation fails with an error, so must the other. It is not possible to use the message-not-understood error to gain information about the input.

6. Inference

It is possible to adopt the work of Palsberg [19] for type inference [3]. The inference algorithm is constraint-based, and works by reducing constraints first to a graph and then an automaton. The language accepted by the automaton for different start states describes the types of the corresponding expressions.

Because of our changed typing rules, and most significantly the change in the subtyping relationship, compared to the original object calculus, a modification of the constraint rules is necessary. The adapted rules are given in Figure 6, where changes to Palsberg's work have been underlined. With those changes, it is easy to prove that a system of constraints generated by an expression has a solution if and only if the expression is typeable, and the solution for the expression is a valid type (cf. Lemma 4.2 in [19]).

[3] In our context, typing, since we do not use explicit method parameter annotations.

x is a variable:
$$x \leq [\![x]\!]$$

$[m_i = \varsigma(x_i)o_i]$ is a literal:
$$[m_i : [\![o_i]\!]] \leq [\![[m_i = \varsigma(x_i)o_i]]\!]$$
$$x_i = \underline{[\![[m_i = \varsigma(x_i)o_i]]\!]}$$

$o.m_i$ is invocation:
$$[m_i : \langle o.m_i \rangle] \,\underline{\leq}\, [\![o]\!]$$
$$\langle o.m_i \rangle \leq [\![o.m_i]\!]$$

$o \leftarrow m_i = \varsigma(x_i)o_i$ is method extension/override:
$$[\![o]\!] \leq [\![o \leftarrow m_i = \varsigma(x_i)o_i]\!]$$
$$[m_i : [\![o_i]\!]] \,\underline{\leq}\, [\![o]\!]$$
$$[\![o]\!] = x$$

Figure 6. Constraint Rules

The system of constraints is then translated into an equivalent graph representation. Types and expressions become nodes. Constraints establish directed $\leq$-labeled edges, while types also introduce method-labeled edges. The graph is closed for $\leq$ edges, that is, edges for reflexivity and transitivity of $\leq$ are added. Furthermore, since the type system only supports width-subtyping, edges are included to enforce this property[4]. A solution of the graph is a mapping of variable nodes to types, such that the subtyping on edges is satisfied. It is obvious that a solution to a constraint system exists if and only if a solution to the corresponding graph exists (cf. Theorem 5.2 in [19]).

Finally, the graph is used to derive an automaton with the method names as alphabet and states corresponding to the nodes of the graph. All states are accepting. The automaton has transitions between states with ϵ if there's a $\leq$-labelled edge, and with the name of the method on a method-labelled edge. However, since our subtyping relation is inverse to that of Abadi and Cardelli, and the typing rules usually establish lower bounds, we inverse the direction of the $\leq$-edges, e.g., a constraint $a \leq b$ induces an $a \xleftarrow{\leq} b$, which defines a transition $b \xrightarrow{\epsilon} a$.

The language $\mathcal{L}(s)$ is defined as all the words accepted when starting from state s. It can be shown that $\mathcal{L}$ is a solution for the graph (cf. Theorem 6.5 in [19]). The proofs in [19] have to be adapted for the reversed subtyping relationship, but are structurally the same. If the language $\mathcal{L}$ is finite for the top-level expression, the types are finite and can be expressed in our system. Otherwise, recursive types are necessary.

From this basic inference of the structure of types, we can iteratively propagate labels from free variables, which need to be defined in a type environment, to obtain a complete typing. All types are initially assumed to be low. If a type mapping contains a high label, it is joined, into the enclosing expression according to the type rules. Then the process is repeated, until either the propagation finishes or a contradiction is found.

7. Related Work

For a good survey of language-based information flow control, we refer to [22]. Several treatments of information flow in object-oriented languages have been presented, e.g., [6, 18, 21, 23]. These are all class-based, a quite different setting from ours, with its own challenges and simplifications.

Our proofs were inspired by work of Barthe and Serpette [7]. To the best of our knowledge, thiss is the only other information flow

[4] Note that the closure is changed in direction to account for our subtyping.

work in foundational object calculi. The work presents an annotated version of Abadi and Cardelli's object calculus and its first-order type system. Our work differs in that it allows object extension and does not rely on Abadi & Cardelli as an underlying type system, and thus execution might produce method-not-found errors.

Our work is related to [4]. Askarov and Sabelfeld describe an extension of JIF that allows to designate exceptions that cannot be caught and lead to termination. While on the surface similar, goals and features are quite different. Our work considers object-oriented programming based around objects and allows adding methods to objects, a feature widely used in dynamic languages, while an extension of JIF is class-based and thus objects are static.

Most work for information flow control in dynamic languages is non-static. One noteable exception that is related to our work is [12] for Javascript. A flow analysis is performed to conservatively compute influences. The process is staged, since the presence of `eval` in Javascript makes a whole-program analysis unfeasible. To allow a fast syntactic check when loading Javascript, the enforceable policies are very simplistic: a variable may not be read or may not be written. In comparison, we support any security lattice, for example, the very expressive DLM [17], and an annotation checker could be used to verify annotations. The staging process could be integrated in our work to support `eval`.

Our work is partially inspired by previous work on extensible object calculi. Two main calculi have been proposed [1, 14], and both have been extended to include subtyping and object extension [9, 11, 15].

Incomplete objects have been considered in [8, 10]. Types are split up into an interface and a completion component. Only methods in the interface component might be called, while the completion component defines dependencies (e.g., method that need to be added to complete an object). Our type system does not separate between interface and completion components, since we want to allow calling potentially non-existing methods.

8. Conclusion

We have shown how to establish noninterference without the need of a proof of type soundness in the underlying system. The current work is in a first-order functional object calculus with method extension. This simplifies the work significantly. However, ongoing research work is being pursued in the following directions:

For the imperative setting, our simple unified type is not sufficient. In the functional calculus, the second example program from section 2 collapses, because any error in a high context cannot be observed by a low attacker.

We are experimenting with a three-segment object type, inspired by two-segment types in [10, 15]. Methods in the first segment are known to exist. Methods in the third segment are known to be missing. Methods in the middle segment may exist. Operations allow to add methods (second or third to first) and remove methods (first or second to third). Subtyping ensures that we can merge object types by moving methods from first to second and third to second, and add new methods as missing.

The previous encoding would allow us to explicitly remove methods. This is an interesting feature not covered in other calculi (since it is not type-safe when the goal is absence of message-not-found).

Acknowledgements

The authors would like to thank the anonymous reviewers for their comments that substantially improved this paper. This work was supported, in part, by the National Science Foundation under grants CCF-0846010 and CNS-0964710.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[2] Christopher Anderson and Sophia Drossopoulou. δ: an imperative object based calculus. In *USE 2002*, 2002.

[3] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In Sushil Jajodia and Javier Lopez, editors, *Computer Security ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer Berlin / Heidelberg, 2008.

[4] Aslan Askarov and Andrei Sabelfeld. Catch me if you can: permissive yet secure error handling. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 45–57, New York, NY, USA, 2009. ACM.

[5] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.

[6] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *In IEEE Computer Security Foundations Workshop (CSFW*, pages 253–267. IEEE Computer Society Press, 2002.

[7] Gilles Barthe and Bernard P. Serpette. Partial evaluation and noninterference for object calculi. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 53–67, London, UK, 1999. Springer-Verlag.

[8] Lorenzo Bettini, Viviana Bono, and Silvia Likavec. A core calculus of mixins and incomplete objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 208–209, New York, NY, USA, 2004. ACM.

[9] Viviana Bono and Michele Bugliesi. Matching constraints for the lambda calculus of objects. In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications*, pages 46–62, London, UK, 1997. Springer-Verlag.

[10] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping constraints for incomplete objects (extended abstract). In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '97, pages 465–477, London, UK, 1997. Springer-Verlag.

[11] Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 462–497, London, UK, 1998. Springer-Verlag.

[12] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[13] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[14] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. of Computing*, 1:3–37, March 1994.

[15] Luigi Liquori. On object extension. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 498–522, London, UK, 1998. Springer-Verlag.

[16] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[17] A.C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. *IEEE Symposium on Security and Privacy*, 1998.

[18] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[19] Jens Palsberg. Efficient inference of object types. *Inf. Comput.*, 123:198–209, December 1995.

[20] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[21] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 63–74, New York, NY, USA, 2009. ACM.

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

[23] Nikhil Swamy, Juan Chen, and R Chugh. *Enforcing Stateful Authorization and Information Flow Policies in Fine*, volume 6012 of *Programming Languages and Systems*, pages 529–549. Springer, 2010.