From Exponential to Polynomial-time Security Typing via Principal Types

Sebastian Hunt[1] and David Sands[2]

[1] City University London
[2] Chalmers University of Technology, Sweden

Abstract. Hunt and Sands (POPL'06) studied a flow sensitive type (FST) system for multi-level security, parametric in the choice of lattice of security levels. Choosing the powerset of program variables as the security lattice yields a system which was shown to be equivalent to Amtoft and Banerjee's Hoare-style independence logic (SAS'04). Moreover, using the powerset lattice, it was shown how to derive a principal type from which all other types (for all choices of lattice) can be simply derived. Both of these earlier works gave "algorithmic" formulations of the type system/program logic, but both algorithms are of exponential complexity due to the iterative typing of While loops. Later work by Hunt and Sands (ESOP'08) adapted the FST system to provide an erasure type system which determines whether some input is correctly erased at a designated time. This type system is inherently exponential, requiring a double typing of the erasure-labelled input command. In this paper we start by developing the FST work in two key ways: (1) We specialise the FST system to a form which only derives principal types; the resulting type system has a simple algorithmic reading, yielding principal security types in polynomial time. (2) We show how the FST system can be simply extended to check for various degrees of termination sensitivity (the original FST system is completely termination insensitive, while the erasure type system is fully termination sensitive). We go on to demonstrate the power of these techniques by combining them to develop a type system which is shown to correctly implement erasure typing in polynomial time. Principality is used in an essential way to reduce type derivation size from exponential to linear.

1 Introduction

The control of information flow is at the heart of many security goals. The classic multi-level security policy says that if a piece of data is considered secret from the perspective of a certain observer of a system, then during execution of the system there should be no information flow from the datum to that observer. Denning and Denning [DD77] pioneered the use of program analysis to statically determine if the information flow properties of a program satisfy a certain multi-level security policy.

A significant trend in the last 10 years has been the use of *security type systems* to formulate the analysis of secure information flow in programs, and to aid in a rigorous proof of its correctness.

The most well-cited works in this area [DD77, VSI96] are *flow-**in**sensitive*, meaning that a *fixed security level* is associated with each variable or data container. To understand the limitations of flow-insensitivity in this context consider the following code

fragment which swaps the values of two secrets and then swaps the values of two non secrets, via a temporary variable:

```
tmp := secret1; secret1 := secret2; secret2 := tmp;
tmp := public1; public1 := public2; public2 := tmp;
```

The program above is not typeable in a flow-insensitive system because the variable `tmp` cannot be correctly assigned a *single* security level. In a flow sensitive system this obviously secure program becomes typeable because the level of variable `tmp` can vary over time to more accurately reflect the security level of its contents.

Our earlier work [HS06] studies a flow-sensitive type (FST) system for multi-level security, parametric in the choice of lattice of security levels. It is shown that choosing the powerset of program variables as the security lattice yields a system which is equivalent to Amtoft and Banerjee's Hoare-style independence logic [AB04]. Moreover, using the powerset lattice, it is is shown how to derive a *principal typing* from which all other typings (for all choices of lattice) can be simply derived. The FST system is reviewed in Section 2. Later work by Hunt and Sands [HS08] adapts the FST system to provide an *erasure* type system which determines whether some input is correctly erased at a designated time. In this formulation, flow-sensitivity is essential to erasure typing.

The original FST system and the system of Amtoft and Banerjee (including [AB07]) provided "algorithmic" formulations of the type system/program logic, but both algorithms are of exponential complexity due to the iterative typing of While loops.

The erasure type system includes at its core a variant of the FST system. The core differs slightly from the earlier FST system in that it is termination-sensitive (meaning that it does not ignore those dependencies which arise purely from termination behaviour). The key to the erasure typing, however, is the typing of the erasure-labelled input command:

$$\text{input } x : L_1 \text{ erased to } L_2 \text{ after } C$$

This command inputs a value from the channel of security level L_1 and places it in variable x, and then executes command C. The erasure specification "erased to L_2" says that after execution of C the value that was input will only be observable to observers at level L_2 and above. So in particular if level L_2 is sufficiently high (so that there are no observers at all) then the information is completely erased.

To type this command we first establish that C is well typed in a context where x initially has type L_1. But then to deal with the erasure condition we perform a *second* typing in which we assume that x initially has type L_2, and where we ignore the effects of any output statements in C.

From an algorithmic perspective, the erasure type system is more inherently exponential than the underlying FST system, in the sense that the non-algorithmic typing derivations themselves can be exponential in the size of the original program. This is due to a subprogram C being typed twice within the erasure-labelled input rule.

Contributions In this paper we start by developing the FST work in two key ways:

1. We specialise the FST system to a form which *only* derives principal typings (Section 2.2); the resulting type system is compact and simple – arguably simpler than all the previous descriptions of flow-sensitive security analyses – and at the same time admits a direct algorithmic reading, yielding principal security typings in polynomial time (Section 2.4).
2. We show (Section 3) how the FST system can be simply generalised to be parametric in the degree of termination-sensitivity (the original FST system is completely termination-insensitive, while the erasure type system is fully termination-sensitive). From the principal typing of a program we can then deduce both termination-sensitive (i.e. sensitive to information flows transmitted by the termination status of a program) and termination-insensitive information flow properties.

We go on to demonstrate the power of these techniques by combining them to develop a type system which is shown to correctly implement erasure typing in polynomial time (Section 4). Principality is used in an essential way to reduce type derivation size from exponential to linear. Again, the key idea is to specialise the system so that it only derives principal typings. The notion of principality is sufficiently general that the two different typings of the erasure command can be derived cheaply by instantiating a single principal typing.

Finally (Section 5) we sketch how the FST system can be extended to handle recursive procedures in polynomial time. The analysis is polymorphic (and hence context-sensitive) in the procedures, but strikingly the extension to this case does not need to introduce type variables, and the algorithm does not require the introduction of type constraints and constraint solving.

Related work is discussed in Section 6.

2 Flow-sensitive security types

We begin by recalling from [HS06] the algorithmic version of the FST system[3]. See Fig. 1. We refer to this system as *Flow Core* (FC). In the While rule, **fix** denotes the least fixed-point operator. Well definedness depends on the fact that the typing rules define a monotone function (see below).

Notation: we will be defining a number of alternative type systems; in most cases we use an undecorated turnstile ($\vdash$) in judgements, relying on context to clarify which particular type system we mean; where we need to make our intention more explicit (typically, when comparing different systems) we decorate the turnstile with a subscript (for example, writing $\vdash_{\mathrm{FC}}$ for the Flow Core system).

All the type systems we consider are concerned with tracking information flows with respect to various hierarchies of security levels; each such hierarchy consists of a join-semilattice with a least element (that is, a partial order in which all finite sets have a least upper bound). Wherever we say join-semilattice in the sequel we mean one with a least element; we overload $\bot$ to mean the least element of whatever join-semilattice is

[3] There are some notational differences from the original presentation.

$$\text{Skip}\frac{}{p \vdash \Gamma\ \{\mathbf{skip}\}\ \Gamma} \qquad \text{Assign}\frac{a = \Gamma(E)}{p \vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup a]}$$

$$\text{Seq}\frac{p \vdash \Gamma\ \{C_1\}\ \Gamma_1 \quad p \vdash \Gamma_1\ \{C_2\}\ \Gamma_2}{p \vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma_2}$$

$$\text{If}\frac{a = \Gamma(E) \quad p \sqcup a \vdash \Gamma\ \{C_i\}\ \Gamma_i \quad i = 1, 2}{p \vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ (\Gamma_1 \sqcup \Gamma_2)}$$

$$\text{While}\frac{\Gamma_f = \mathbf{fix}(\lambda\Gamma.\,\mathbf{let}\ p \sqcup \Gamma(E) \vdash \Gamma\ \{C\}\ \Gamma'\ \mathbf{in}\ \Gamma' \sqcup \Gamma_0)}{p \vdash \Gamma_0\ \{\mathbf{while}\ E\ C\}\ \Gamma_f}$$

Fig. 1. Flow Core (FC)

under discussion. Each security level models a class of users, grouped according to how much they are permitted to observe; if $a \sqsubseteq b$ then b-users can see everything a-users can see, and maybe more. Let $\mathcal{L}$ be a join-semilattice. Typing judgements in Flow Core have the form

$$p \vdash \Gamma\ \{C\}\ \Gamma'$$

where C is a command, $p \in \mathcal{L}$ represents the security level of the "program counter", and $\Gamma, \Gamma' : PVar \to \mathcal{L}$ are environments mapping program variables to security levels. ($PVar$ is the finite set of variables used in whatever top-level program is being analysed. Formally this should be an explicit parameter of the typing judgement, but glossing over this detail saves us from notational clutter without causing any significant problems.) Throughout the paper we treat function spaces such as $PVar \to \mathcal{L}$ as join-semilattices, inheriting their lattice structure pointwise from $\mathcal{L}$ (so $\Gamma \sqcup \Gamma' = \lambda x.\Gamma(x) \sqcup \Gamma'(x)$, etc). In these rules, $\Gamma(E)$ means the join of the levels assigned to the free variables in the expression E:

$$\Gamma(E) = \bigsqcup_{x \in \mathrm{fv}(E)} \Gamma(x)$$

When E has *no* free variables (it only involves constants) then $\Gamma(E) = \bigsqcup\{\} = \bot$. We leave the syntax of expressions unspecified; semantic soundness assumes that expressions are free of side-effects and are interpreted as total functions of the values of their free variables.

Note that $\mathcal{L}$ is an implicit parameter in the definition of Flow Core, so Flow Core actually defines a *family* of type systems, indexed by the choice of $\mathcal{L}$.

Although the current paper is not primarily concerned with semantic soundness, it is helpful to have some intuitions. By assigning levels to variables, an environment determines a policy, for each level $a \in \mathcal{L}$, stating which parts of a memory state a-users should be allowed to see. Say that two memory states are *a-equivalent under policy Γ* iff they agree on all variables x such that $\Gamma(x) \sqsubseteq a$. Then the intended semantics of a judgement $p \vdash \Gamma\ \{C\}\ \Gamma'$ is that it should satisfy

1. C only changes variables x for which $\Gamma'(x) \sqsupseteq p$; and
2. for all a, if two initial memory states are a-equivalent under Γ, the corresponding final stores after execution of C will be a-equivalent under Γ'.

Semantic soundness of Flow Core with respect to this specification is proved in [HS06].

The use of two type environments (pre- and post-) in Flow Core deserves some explanation. For a top-level program executed in batch mode, it is perhaps more natural for a security policy to specify a single assignment of levels to variables, applying both before and after the program executes. But it is the use of two distinct environments which allows Flow Core to be flow-sensitive. Consider again the swap program from the introduction:

```
tmp := secret1; secret1 := secret2; secret2 := tmp;
tmp := public1; public1 := public2; public2 := tmp;
```

It is a simple exercise to verify that this can be typed in Flow Core with $\mathit{low} \vdash \Gamma\ \{P\}\ \Gamma$ for Γ mapping `tmp` and the public variables to *low*, and mapping the secret variables to *high*. The first assignment raises the level of `tmp` from *low* to *high*; this is modelled by fact that the post-environment for the corresponding sub-derivation is not Γ but $\Gamma' = \Gamma[\texttt{tmp} \mapsto \mathit{high}]$. The assignment `tmp := public1` later reduces the level back down to *low*; this is modelled by the fact that the post-environment for the corresponding sub-derivation is $\Gamma'[\texttt{tmp} \mapsto \mathit{low}] = \Gamma$. In general, to enforce a policy specified by a single Γ, we would compute the typing $\bot \vdash \Gamma\ \{P\}\ \Gamma'$ and then check that $\Gamma' \sqsubseteq \Gamma$ (it is safe for a program to make variables *less* informative than the policy allows). In a language with IO channels, we are more likely to require a fixed policy for channels and assume that program variables are not directly observable at all. This scenario can also be modelled using essentially the same approach (see Section 4).

Flow Core is described in [HS06] as "algorithmic", by which we mean that the rules are syntax directed and that, for a given choice of p, Γ, they determine exactly one derivation for each C. A consequence is that Flow Core is *functional*: for each command C, for all p, Γ, there exists a unique Γ' such that $p \vdash \Gamma\ \{C\}\ \Gamma'$ (it is also monotonic in p and Γ). In [HS06] finite convergence of the While rule's fixed-point construction was trivially guaranteed by the requirement that $\mathcal{L}$ should be finite; here we have relaxed that constraint by requiring only that $\mathcal{L}$ should have finite joins. Nonetheless, finite convergence is guaranteed because environments are finite and the typing rules only ever construct elements of $\mathcal{L}$ which are finite joins of lattice elements actually used in the initial environment. Note that the use of **fix** makes the typing of nested while loops exponential, since the body of a loop will be typed repeatedly, once for each iteration towards the fixed-point.

2.1 FST with a pc-variable

The principal typings result presented in [HS06] is slightly less general than we would like (and than we need in what follows) because it is restricted to "top-level" typings, by which we mean typings where p is $\bot$. To generalise the result we make a small change to Flow Core: we adjoin a new variable $\mathbf{pc}$ to model the program counter and we track $\mathbf{pc}$ in the type environments, along with the program variables, rather than reserving a special place for it in the syntax of judgements. The slight disadvantage is that the If and While rules must now explicitly "reset" $\mathbf{pc}$ in their post-environments. The advantage is that we can easily state a fully general principal typings result, paving

the way to the key contributions of the current paper. Let $Var = PVar \cup \{\mathbf{pc}\}$. Type environments are extended to this new domain, thus $\Gamma : Var \to \mathcal{L}$. Except for the "resets" mentioned previously, when $\mathbf{pc}$ is updated in the new type rules it is always *increased*. It is convenient to introduce some notation for such increasing updates: we write $\Gamma[x \mathrel{+}= a]$ to mean $\Gamma[x \mapsto \Gamma(x) \sqcup a]$. We refer to the modified system as *Flow Core-pc* (Fpc). The rules are presented in Fig. 2.

$$\text{Skip}\frac{}{\vdash \Gamma\ \{\mathbf{skip}\}\ \Gamma} \qquad \text{Assign}\frac{p = \Gamma(\mathbf{pc}) \quad a = \Gamma(E)}{\vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup a]}$$

$$\text{Seq}\frac{\vdash \Gamma\ \{C_1\}\ \Gamma_1 \quad \vdash \Gamma_1\ \{C_2\}\ \Gamma_2}{\vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma_2}$$

$$\text{If}\frac{p = \Gamma(\mathbf{pc}) \quad a = \Gamma(E) \quad \vdash \Gamma[\mathbf{pc} \mathrel{+}= a]\ \{C_i\}\ \Gamma_i \quad i = 1, 2}{\vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ (\Gamma_1 \sqcup \Gamma_2)[\mathbf{pc} \mapsto p]}$$

$$\text{While}\frac{p = \Gamma(\mathbf{pc}) \quad \Gamma_f = \mathbf{fix}(\lambda\Gamma.\, \mathbf{let}\ \vdash \Gamma[\mathbf{pc} \mathrel{+}= \Gamma(E)]\ \{C\}\ \Gamma'\ \mathbf{in}\ \Gamma' \sqcup \Gamma_0)}{\vdash \Gamma_0\ \{\mathbf{while}\ E\ C\}\ \Gamma_f[\mathbf{pc} \mapsto p]}$$

Fig. 2. Flow Core-pc (Fpc)

It is easily checked that derivations in Flow Core-pc preserve the type assignment for $\mathbf{pc}$:

Lemma 1. *If* $\vdash \Gamma\ \{C\}\ \Gamma'$ *then* $\Gamma'(\mathbf{pc}) = \Gamma(\mathbf{pc})$.

A straightforward induction on C then establishes that Flow Core-pc is essentially equivalent to Flow Core:

Theorem 1. $p \vdash_{\mathrm{FC}} \Gamma\ \{C\}\ \Gamma'$ *iff* $\vdash_{\mathrm{Fpc}} \Gamma[\mathbf{pc} \mapsto p]\ \{C\}\ \Gamma'[\mathbf{pc} \mapsto p]$

Note that while the theorem appears to leave open the possibility that there may be Flow Core-pc derivations $\vdash \Gamma\ \{C\}\ \Gamma'$ without any counterpart in Flow Core, this is ruled out by Lemma 1.

2.2 Principal Typings

Intuitively, a *principal typing* for a term is a typing from which all its other typings may be simply recovered. In this section we show that every command does indeed have a principal Flow Core-pc typing in this sense. More formally, the most general definition of principal typing is due to Wells [Wel02]. We do not use Wells' definition in the current paper but it is a simple corollary of Theorem 2 (see below) that our principal typings are indeed principal according to that definition.

The following lemma establishes a property of Flow Core-pc which is fundamental to the rest of the technical development. Say that a map α is *join-preserving* if it preserves all finite joins or, equivalently, if $\alpha(\bot) = \bot$ and $\alpha(a \sqcup b) = \alpha(a) \sqcup \alpha(b)$. It turns out that join-preserving maps allow us freely to translate one valid typing into another.

Lemma 2. *Let $\mathcal{L}, \mathcal{J}$ be join-semilattices, let $\Gamma, \Gamma' : Var \to \mathcal{L}$ and let $\alpha : \mathcal{L} \to \mathcal{J}$ be join-preserving. If $\vdash \Gamma \ \{C\} \ \Gamma'$ then $\vdash \alpha \circ \Gamma \ \{C\} \ \alpha \circ \Gamma'$.*

The proof of this lemma essentially relies only on the fact that environment update and $\sqcup$ are the key operations used in the type rules. This also holds for all the other algorithmic type systems presented in the current paper and so the lemma easily extends to them. We rely on this fact without further comment in the sequel.

Principal typings are constructed by choosing $\mathcal{P}(Var)$ as the security lattice. In what follows we let Δ range over type environments just in this case (thus $\Delta : Var \to \mathcal{P}(Var)$). The semantic content of such a Δ can be understood simply as a set of dependencies: $y \in \Delta(x)$ means "y depends on x". On this reading, it would be very natural to represent environments directly as binary relations rather than functions, so that we could write, for example, $x \ \Delta \ y$ in place of $y \in \Delta(x)$. However, using a different representation just when $\mathcal{L} = \mathcal{P}(Var)$ would make it awkward to relate $\mathcal{P}(Var)$ typings to typings for other choices of $\mathcal{L}$, which is necessary to establish our principal typings result. To square this circle we introduce a notion of *monadic composition*.

For any finite set B, join-semilattice $\mathcal{L}$ and $g : B \to \mathcal{L}$, define $g^\dagger : \mathcal{P}(B) \to \mathcal{L}$ by $g^\dagger(X) = \bigsqcup_{x \in X} g(x)$. In much of what follows, $\mathcal{L}$ will itself be a powerset, in which case it is implicit that $\sqcup$ is $\cup$. Note that, for any $g : B \to \mathcal{L}$, $g^\dagger$ is join-preserving: $g^\dagger(X \cup Y) = g^\dagger(X) \sqcup g^\dagger(Y)$ and $g^\dagger(\{\}) = \bot$. Now, given $f : A \to \mathcal{P}(B)$ and $g : B \to \mathcal{L}$, define the monadic composition $f \ ; g : A \to \mathcal{L}$ by $f \ ; g = g^\dagger \circ f$. Note that given two type environments $\Delta, \Delta' : Var \to \mathcal{P}(Var)$, the relational reading of $\Delta \ ; \Delta'$ is simply relational composition (abusing notation, $x \ \Delta \ ; \Delta' \ y$ iff $\exists z. x \ \Delta \ z \wedge z \ \Delta' \ y$).

Lemma 3 (Kleisli Axioms). *For each finite set A, let $\eta_A : A \to \mathcal{P}(A)$ be the map $x \mapsto \{x\}$. Then:*

1. $\eta_A{}^\dagger = \mathrm{id}_{\mathcal{P}(A)}$
2. $\eta_A \ ; f = f$ *for* $f : A \to \mathcal{L}$
3. $f^\dagger \ ; g = (f \ ; g)^\dagger$ *for* $f : A \to \mathcal{P}(B)$ *and* $g : B \to \mathcal{L}$

If we restrict attention to the cases where $\mathcal{L}$ is a powerset, these are exactly the Kleisli axioms for the canonical powerset monad (see, for example, [Mog89]).

Note that η_{Var} (as defined in Lemma 3) is the environment $\lambda x \in Var. \{x\}$. Henceforth we just write η for η_{Var}. The relational reading of η is just the identity relation.

Let $\vdash \eta \ \{C\} \ \Delta$ (since Flow Core-pc is functional, such a typing exists and is uniquely determined by C). It is this typing which we claim as the principal typing for C. The following theorem justifies the claim by showing how every other typing for C can be recovered from this one:

Theorem 2 (Principal Typings). *Let $\vdash \eta \ \{C\} \ \Delta$. Then $\vdash \Gamma \ \{C\} \ \Gamma'$ iff $\Gamma' = \Delta \ ; \Gamma$.*

Proof. Since the type system is functional, it suffices to show that $\vdash \Gamma \ \{C\} \ \Delta \ ; \Gamma$. By definition of monadic composition, $\Delta \ ; \Gamma = \Gamma^\dagger \circ \Delta$. Since $\Gamma^\dagger$ preserves unions, Lemma 2 says that $\vdash \Gamma^\dagger \circ \eta \ \{C\} \ \Gamma^\dagger \circ \Delta$ or, equivalently, $\vdash \eta \ ; \Gamma \ \{C\} \ \Delta \ ; \Gamma$. By the second Kleisli axiom, $\eta \ ; \Gamma = \Gamma$, thus $\vdash \Gamma \ \{C\} \ \Delta \ ; \Gamma$. □

From now on we refer to Δ simply as the principal *type* for C (this is the only part of the principal typing which is specific to C, since the other component is always η).

2.3 The Principal Type System

We formulate a type system which *only* constructs principal types and in which every sub-derivation also derives a principal type. This system is arrived at by specialising Flow Core-pc to the case that $\mathcal{L} = \mathcal{P}(\mathit{Var})$ and $\Gamma = \eta$ and using the Principal Typings Theorem to replace each sub-derivation by a principal type derivation. Crucially, this allows us to replace the multiple sub-derivations required in the While rule by a single derivation, thus removing the exponential cost which they incur. There is just one eureka step in the specialisation of the While rule; we implement the fixed-point construction as a transitive closure. We write Δ^* for the *reflexive-transitive closure* of Δ, defined in the standard way but transposed into functional form:

$$\Delta^* = \bigsqcup_{n \geq 0} \Delta^n$$

where $\Delta^0 = \eta$ and $\Delta^{n+1} = \Delta \mathbin{;} \Delta^n$. (Note that the "infinite join" in this definition will actually be finite, since Var is finite, though even without this constraint it would be well-defined, since $\mathit{Var} \to \mathcal{P}(\mathit{Var})$ would still be a complete lattice.)

$$\text{Skip}\frac{}{\vdash \mathbf{skip} : \eta} \qquad \text{Assign}\frac{}{\vdash x := E : \eta[x \mapsto \{\mathbf{pc}\} \cup \mathrm{fv}(E)]}$$

$$\text{Seq}\frac{\vdash C_i : \Delta_i \quad i = 1,2}{\vdash C_1 \mathbin{;} C_2 : \Delta_2 \mathbin{;} \Delta_1}$$

$$\text{If}\frac{\vdash C_i : \Delta_i \quad \Delta_i' = \Delta_i \mathbin{;} \eta[\mathbf{pc} \mathrel{+}= \mathrm{fv}(E)] \quad i = 1,2}{\vdash \mathbf{if}\ E\ C_1\ C_2 : (\Delta_1' \sqcup \Delta_2')[\mathbf{pc} \mapsto \{\mathbf{pc}\}]}$$

$$\text{While}\frac{\vdash C : \Delta \quad \Delta_f = (\Delta \mathbin{;} \eta[\mathbf{pc} \mathrel{+}= \mathrm{fv}(E)])^*}{\vdash \mathbf{while}\ E\ C : \Delta_f[\mathbf{pc} \mapsto \{\mathbf{pc}\}]}$$

Fig. 3. Flow Principal (FP)

Theorem 3. *Flow Principal derives principal types:* $\vdash_{\mathrm{Fpc}} \eta\ \{C\}\ \Delta$ *iff* $\vdash_{\mathrm{FP}} C : \Delta$.

Proof sketch: Proof is by a simple induction on the structure of C showing that each Flow Principal rule is the specialisation of its Flow Core-pc counterpart, using the Principal Typings Theorem to replace sub-derivations with principal type derivations. For example, consider the Seq rule. The Flow Core-pc derivation is:

$$\text{Seq}\frac{\vdash \eta\ \{C_1\}\ \Delta_1 \quad \vdash \Delta_1\ \{C_2\}\ \Delta_2'}{\vdash \eta\ \{C_1 \mathbin{;} C_2\}\ \Delta_2'}$$

The derivation for C_2 is not principal, so we replace it by $\vdash \eta\ \{C_2\}\ \Delta_2$ and then apply the Principal Typings Theorem to derive $\Delta_2' = \Delta_2 \mathbin{;} \Delta_1$.

The proof for the While rule is slightly more involved, because we have to show that the reflexive-transitive closure in the Flow Principal rule correctly implements the fixed-point specified in the Flow Core-pc rule, but in essence we are able to equate each term in an ascending chain for the transitive closure with the corresponding term in the fixed-point chain.

2.4 Complexity

Flow Principal (Fig. 3) has a direct algorithmic reading. Here we sketch the complexity based on a direct implementation of a relational reading of the rules.

The key operations that must be implemented to construct a type are composition (;), update ($[\cdot \mapsto \cdot]$), union, and reflexive-transitive closure ($*$). Representing binary relations as boolean matrices has a long tradition in program analysis (see e.g. [Pro59]). In this representation we have one row for each element of the domain and one column for each element of the range. Thus in the case of our relations between variables, Δ is represented by a matrix for which there is a 1 in the row for x and column for y if and only if $x \ \Delta \ y$. Composition is then realised by boolean matrix multiplication, union is just boolean matrix addition (pointwise conjunction), and the single-value update operation is just row replacement. Using this representation we can easily construct a polynomial time complexity for type inference:

Theorem 4. *Flow Principal can be used to construct principal types in $O(nv^3)$ where n is the size of the program and v is the number of variables.*

Proof. Since the size of a type derivation is $O(n)$ there are thus $O(n)$ operations required to construct the type. Considering the cost of the operations, the potentially expensive operations are composition and reflexive-transitive closure. Adopting the boolean matrix representation, the matrices have size v^2, and thus the cost of composition (matrix multiplication) is v^3. Using Warshall's algorithm the cost of transitive (and reflexive) closure is also v^3. Hence the total cost of constructing the principal type is $O(nv^3)$.

3 Termination typing

The FC system, like Denning and Denning's original flow-insensitive analysis of secure information, enforces an imperfect notion of information flow which has become known as *termination-insensitive noninterference*. Under this version of noninterference, information leaks are permitted if they are transmitted purely by the program's termination behaviour (i.e., whether it terminates or not). This imperfection is the price to pay for having a security condition which is relatively liberal (e.g. allowing while-loops whose termination may depend on the value of a secret) and easy to check.

But in some circumstances (for example in the presence of IO [AHSS08]) it may be desirable to enforce a stronger condition such as termination-sensitive security in which a secret is not permitted to transmit information even through termination. For example, suppose that h is high and l is low, then the following programs are not termination-sensitive secure:

```
if h then (while true skip) else skip; l := 1
```

```
(while h skip) ; l := 1
```

(An example of a system enforcing this stronger condition is [AB04].) In what follows we generalise the FST system to track the *degree* of termination-sensitivity. This

provides a variant of the system from which a principal type provides both termination-sensitive and termination-insensitive typings, and even typings which lie between the two [DS09], for example where we can abide some termination leakage from a large piece of data (on the basis that the rate of leakage is low) but not for small data.

The idea is to add a termination variable $\mathbf{t}$ to the state to record the levels upon which termination of the command may depend. This corresponds to the *termination effect* from [Bou05] (a similar component can be found in earlier type systems [Smi01, BC01]).[4] The only rule which needs to consider the value of the termination variable is the While-rule. Consider the While rule in Fig. 2. The fixed-point constructs an environment Γ_f in which $\mathbf{pc}$ records the maximum level of data which can influence the value of the loop condition. We therefore need to modify the rule to ensure that it raises $\mathbf{t}$ to this level. Modifying the While rule in this way gives us a new type system which we refer to as *Flow Termination* (FT). See Fig. 4. It is clear by comparing the

$$\text{While}\frac{p = \Gamma(\mathbf{pc}) \quad \Gamma_f = \mathbf{fix}(\lambda\Gamma.\,\mathbf{let}\ \vdash \Gamma[\mathbf{pc} \mathrel{+}= \Gamma(E)]\ \{C\}\ \Gamma'\ \mathbf{in}\ \Gamma' \sqcup \Gamma_0)}{\vdash \Gamma_0\ \{\mathbf{while}\ E\ C\}\ \Gamma_f[\mathbf{pc} \mapsto p, \mathbf{t} \mathrel{+}= \Gamma_f(\mathbf{pc})]}$$

Fig. 4. Flow Termination (FT) (*extends Flow Core-pc*)

two versions of the While rule that Flow Termination is functional and is a conservative extension of Flow Core-pc.

After typing a program, the type of the termination variable can be used to determine whether a certain degree of termination-sensitivity has been achieved. To have termination-sensitivity we must demand that the type of $\mathbf{t}$ is $\bot$ - i.e. termination depends only on public inputs. For termination-insensitivity one can simply ignore $\mathbf{t}$. This generalisation to include termination-sensitivity is required for the erasure types considered in the next section.

Although we do not focus on the semantic soundness of the systems in this paper, it is useful to understand the semantic content of the termination variable. We can state it informally as follows: suppose $\vdash_{\text{FT}} \Gamma\ \{C\}\ \Gamma'$ and $a = \Gamma'(\mathbf{t})$. Now suppose that M and N are two memory states (mappings from variables to values) which are a-equivalent under policy Γ. Then C terminates starting with memory state M iff it terminates starting with memory state N.

The Principal Typings Theorem also holds for the Flow Termination system:

Theorem 5. *Let* $\vdash_{\text{FT}} \eta\ \{C\}\ \Delta$. *Then* $\vdash_{\text{FT}} \Gamma\ \{C\}\ \Gamma'$ *iff* $\Gamma' = \Delta\ ;\ \Gamma$.

Specialising the new While rule in the obvious way, we obtain a modified version of Flow Principal which computes principal types for Flow Termination. We refer to this system as *Termination Principal* (TP). See Fig. 5.

Theorem 6. TP *derives principal types:* $\vdash_{\text{FT}} \eta\ \{C\}\ \Delta$ *iff* $\vdash_{\text{TP}} C : \Delta$.

[4] We do not attempt to make the system more liberal in the manner of [Bou05, Smi01, BC01]; this is possible but would require a more pervasive change whereby the type of a variable is a pair of levels.

$$\text{While}\frac{\vdash C : \Delta \quad \Delta_f = (\Delta\ ;\ \eta[\mathbf{pc}\ +\!\!=\ \mathrm{fv}(E)])^*}{\vdash \mathbf{while}\ E\ C : \Delta_f[\mathbf{pc} \mapsto \{\mathbf{pc}\}, \mathbf{t}\ +\!\!=\ \Delta_f(\mathbf{pc})]}$$

Fig. 5. Termination Principal (TP) (*extends Flow Principal*)

4 Erasure types

In this section we derive a polynomial time implementation of the erasure type system from [HS08]. This system uses flow-sensitivity in an essential way to enforce a certain kind of data erasure policy. As explained in the introduction, the system appears to be inherently exponential because it makes essential use of a double typing of the body of the erasure-labelled input command (the Erase rule in Fig.6). Even so, because the core of the erasure type system is an FST system of essentially the same kind as the one from [HS06], we are able to apply the techniques from Sections 2 and 3 above. This enables us to implement erasure typing by transforming the original system into a polynomial-time system for deriving principal types.

4.1 Non-Algorithmic Erasure Type System

Figure 6 presents the type system from [HS08]. Here the syntax $\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C$ abbreviates the construction "input $x : a$ erased to b after C" discussed in the Introduction. We refer to this system as *Erasure Basic* (EB). In rule Erase, deleteOutput(C)

$$\text{Skip}\frac{}{p \vdash \Gamma\ \{\mathbf{skip}\}\ \Gamma} \qquad \text{Assign}\frac{a = \Gamma(E)}{p \vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup a]}$$

$$\text{Erase}\frac{p \vdash \Gamma[x \mapsto a]\ \{C\}\ \Gamma' \quad p \vdash \Gamma[x \mapsto b]\ \{C'\}\ \Gamma' \quad p \sqsubseteq a \quad C' = \text{deleteOutput}(C)}{p \vdash \Gamma\ \{\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C\}\ \Gamma'}$$

$$\text{Output}\frac{b = \Gamma(E) \qquad p \sqcup b \sqsubseteq a}{p \vdash \Gamma\ \{\mathbf{output}\ E\ \mathbf{on}\ a\}\ \Gamma}$$

$$\text{Seq}\frac{p \vdash \Gamma\ \{C_1\}\ \Gamma' \quad p \vdash \Gamma'\ \{C_2\}\ \Gamma''}{p \vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma''} \qquad \text{If}\frac{a = \Gamma(E) \quad p \sqcup a \vdash \Gamma\ \{C_i\}\ \Gamma' \quad i = 1, 2}{p \vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ \Gamma'}$$

$$\text{While}\frac{\Gamma(E) = \bot \quad \bot \vdash \Gamma\ \{C\}\ \Gamma}{\bot \vdash \Gamma\ \{\mathbf{while}\ E\ C\}\ \Gamma} \qquad \text{Sub}\frac{p_1 \vdash \Gamma_1\ \{C\}\ \Gamma_1'}{p_2 \vdash \Gamma_2\ \{C\}\ \Gamma_2'} \quad p_2 \sqsubseteq p_1, \Gamma_2 \sqsubseteq \Gamma_1, \Gamma_1' \sqsubseteq \Gamma_2'$$

Fig. 6. Erasure Basic (EB)

operates on the syntax of C, producing a copy of C in which each $\mathbf{output}$ command has been replaced by $\mathbf{skip}$. Thus to type an erasing input command we have to type its body, C, *twice*, under two different type environments. The first typing enforces non-interference with respect to the security level of the input-channel, while the second typing enforces non-interference with respect to the erasure level; $\mathbf{output}$ commands in C are ignored during the second typing because the erasure level only applies *after* C has executed.

Note that this type system is non-algorithmic. Additionally, in contrast to the FST systems above, it is *not* parametric in the choice of security lattice; input and output commands refer (independently of the type system) to elements from some lattice, which means that programs can only be typed with respect to that particular lattice. Both these issues are addressed in Section 4.2 below.

In this system we are primarily concerned with information flows on the I/O channels. The (flow-sensitive) typing of program variables is thus only a means to the end of checking that the I/O flows comply with the required security policy. Since the type system is monotone in Γ, if a program can be typed at all, it can be typed for the smallest Γ. We say that a command C is *typeable* iff there exists a Γ such that $\bot \vdash \bot \{C\} \Gamma$.

In the following sections we proceed as follows:

1. We develop an algorithmic version of Erasure Basic. At the same time, we generalise the system by introducing auxiliary variables to model the program counter, the termination channel and the IO channels. This allows all constraints to be removed from the type system (rules Erase, Output and While) and makes it parametric in the choice of security type lattice. We establish correctness of the new system with respect to Erasure Basic.
2. We formulate and prove a principal typings result for the generalised system.
3. We specialise the generalised system to one which produces just principal types. This principal types system has the same complexity as the earlier one (Section 2.4).

4.2 Generalised Erasure Type System

We introduce the following additional variables (assumed disjoint from the program variables $PVar$):

- The variables $\mathbf{pc}$ and $\mathbf{t}$, playing the same roles as in Flow Core-pc and Flow Termination above (Sections 2.1 and 3).
- Disjoint sets $IVar$ and $OVar$ of *channel variables*.

We modify the language syntax to use channel variables in place of fixed channel names. Input commands now have the form $\mathbf{input}\ x : i_1 \nearrow i_2\ \mathbf{in}\ C$, with $i_1, i_2 \in IVar$. Output commands have the form $\mathbf{output}\ E\ \mathbf{on}\ o$, with $o \in OVar$. These are the only places in the syntax where channel variables appear. Commands written in the original syntax are encoded in the new syntax by application of injective maps $a \mapsto i_a : \mathcal{L} \to IVar$ and $a \mapsto o_a : \mathcal{L} \to OVar$. For a command C in the old syntax, we denote its encoding in the new syntax by $\widehat{C}$. Thus $\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C$ becomes $\mathbf{input}\ x : i_a \nearrow i_b\ \mathbf{in}\ \widehat{C}$ and $\mathbf{output}\ E\ \mathbf{on}\ a$ becomes $\mathbf{output}\ E\ \mathbf{on}\ o_a$. In the rest of the paper C denotes a command in the new syntax unless explicitly stated otherwise.

Environments become maps $\Gamma : Var \to \mathcal{L}$, where $Var = PVar \cup \{\mathbf{pc}, \mathbf{t}\} \cup IVar \cup OVar$. The rules for the generalised system are presented in Fig. 7. We refer to this system as *Erasure General* (EG).

Some intuitions for the Erasure General rules:

- Once information has flowed to a channel, it has "escaped" the system. This is reflected in the rules by ensuring that, as for $\mathbf{t}$, the post-assignment for a channel variable is always at least as great as its pre-assignment.

$$\text{Skip}\frac{}{\vdash \Gamma\ \{\mathbf{skip}\}\ \Gamma} \qquad \text{Assign}\frac{p = \Gamma(\mathbf{pc}) \quad a = \Gamma(E)}{\vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup a]}$$

$$\text{Erase}\frac{p = \Gamma(\mathbf{pc}) \quad a_j = \Gamma(i_j) \quad \vdash \Gamma[x \mapsto p \sqcup a_j]\ \{C\}\ \Gamma_j \quad j = 1,2 \qquad \forall y \in \mathit{Var}.\Gamma'(y) = \begin{cases} \Gamma_1(y) & \text{if } y \in \mathit{OVar} \\ \Gamma_1(y) \sqcup \Gamma_2(y) & \text{otherwise} \end{cases}}{\vdash \Gamma\ \{\mathbf{input}\ x : i_1 \nearrow i_2\ \mathbf{in}\ C\}\ \Gamma'[i_1 \mathrel{+}= p]}$$

$$\text{Output}\frac{p = \Gamma(\mathbf{pc}) \quad b = \Gamma(E)}{\vdash \Gamma\ \{\mathbf{output}\ E\ \mathbf{on}\ o\}\ \Gamma[o \mathrel{+}= p \sqcup b]}$$

$$\text{Seq}\frac{\vdash \Gamma\ \{C_1\}\ \Gamma_1 \quad \vdash \Gamma_1\ \{C_2\}\ \Gamma_2}{\vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma_2}$$

$$\text{If}\frac{p = \Gamma(\mathbf{pc}) \quad a = \Gamma(E) \quad \vdash \Gamma[\mathbf{pc} \mathrel{+}= a]\ \{C_i\}\ \Gamma_i \quad i = 1,2}{\vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ (\Gamma_1 \sqcup \Gamma_2)[\mathbf{pc} \mapsto p]}$$

$$\text{While}\frac{p = \Gamma(\mathbf{pc}) \quad \Gamma_f = \mathbf{fix}(\lambda\Gamma.\,\mathbf{let}\ \vdash \Gamma[\mathbf{pc} \mathrel{+}= \Gamma(E)]\ \{C\}\ \Gamma'\ \mathbf{in}\ \Gamma' \sqcup \Gamma_0)}{\vdash \Gamma_0\ \{\mathbf{while}\ E\ C\}\ \Gamma_f[\mathbf{pc} \mapsto p, \mathbf{t} \mathrel{+}= \Gamma_f(\mathbf{pc})]}$$

Fig. 7. Erasure General (EG)

- In Erasure Basic the Erase rule imposes the constraint that $p \sqsubseteq a$, where a is the (fixed) input channel type. In Erasure General this constraint has been removed but the post-environment is updated ($\Gamma'[i_1 \mathrel{+}= p]$) in a way which effectively allows the constraint to be checked by examining the final result of the type derivation. The $p \sqcup b \sqsubseteq a$ constraint in the Output rule is handled similarly.
- In Erasure Basic the While rule imposes the constraint that $\Gamma(E) = \bot$. Erasure General uses $\mathbf{t}$ to track whatever termination flows there may be, without building in any constraint on what is allowed. Again, flows not permitted by Erasure Basic can be caught by examining the final result.
- The generalised Erase rule still requires a double typing of the command body. But rather than deleting output commands in the second typing, we are able simply to discard those parts of the derived environment relating to output channels. This is achieved by the definition of Γ', which discriminates between output channel variables and others.

Lemma 4. *Erasure General (Fig. 7) is functional and if* $\vdash \Gamma\ \{C\}\ \Gamma'$ *then:*

1. $\Gamma'(\mathbf{pc}) = \Gamma(\mathbf{pc})$
2. *For all* $x \in \{\mathbf{t}\} \cup \mathit{IVar} \cup \mathit{OVar}$, $\Gamma'(x) \sqsupseteq \Gamma(x)$.

The following theorem shows how each Erasure Basic typing can be recovered from a specified Erasure General typing. They key observation is that the Erasure Basic constraints are satisfied iff the termination and channel variable type assignments are invariant in the specified Erasure General typing. Given $\Gamma : \mathit{PVar} \to \mathcal{L}$ and $p \in \mathcal{L}$, let $\widehat{\Gamma}^{\,p} : \mathit{Var} \to \mathcal{L}$ be defined by $\widehat{\Gamma}^{\,p} = \Gamma[\mathbf{pc} \mapsto p, \mathbf{t} \mapsto \bot][i_a \mapsto a, o_a \mapsto a]_{a \in \mathcal{L}}$.

Theorem 7. *Let C be a command in the original syntax and let $\vdash_{\text{EG}} \widehat{\Gamma}^{\,p}\ \{\widehat{C}\}\ \Gamma'$. Then $p \vdash_{\text{EB}} \Gamma\ \{C\}\ \Gamma_1$ iff there exists $\Gamma_2 \sqsubseteq \Gamma_1$ such that $\Gamma' = \widehat{\Gamma_2}^{\,p}$.*

Corollary 1. *Let C be a command in the original syntax and let $\vdash_{\text{EG}} \bot\ \{\widehat{C}\}\ \Gamma$. Then C is typeable in Erasure Basic iff $\Gamma(\mathbf{t}) = \bot$ and $\Gamma(i_a) = a$ and $\Gamma(o_a) = a$ for each $a \in \mathcal{L}$.*

Thus the problem of Erasure Basic typing is reduced to the problem of constructing a specific Erasure General typing and then checking that the levels of the termination and channel variables remain fixed.

4.3 Principal Types for Erasure Typing

The Erasure General system is sufficiently similar to the earlier FST systems that the principal typings result carries over with no significant extra work:

Theorem 8 (Principal Erasure Typings). *Let $\vdash_{\text{EG}} \eta\ \{C\}\ \Delta$ (see Fig. 7). Then $\vdash_{\text{EG}} \Gamma\ \{C\}\ \Gamma'$ iff $\Gamma' = \Delta\,;\,\Gamma$.*

Proof. As for Theorem 2.

Using the Principal Erasure Typings Theorem we can specialise Erasure General to derive a system which produces only principal erasure types, just as we did in Section 2.2 for the FST system. The derived rules are shown in Fig. 8. We refer to this system as *Erasure Principal* (EP).

$$\text{Erase}\ \frac{\vdash C : \Delta \quad \Delta_j = \Delta\,;\,\eta[x \mapsto \{\mathbf{pc}, i_j\}] \quad j = 1,2 \qquad \Delta'(x) = \begin{cases} \Delta_1(x) & \text{if } x \in \mathit{OVar} \\ \Delta_1(x) \sqcup \Delta_2(x) & \text{otherwise} \end{cases}}{\vdash \mathbf{input}\ x : i_1 \nearrow i_2\ \mathbf{in}\ C : \Delta'[i_1 \mathrel{+}= \{\mathbf{pc}\}]}$$

$$\text{Output}\ \frac{}{\vdash \mathbf{output}\ E\ \mathbf{on}\ o : \eta[o \mathrel{+}= \{\mathbf{pc}\} \cup \mathrm{fv}(E)]}$$

Fig. 8. Erasure Principal (EP) (*extends Termination Principal*)

Theorem 9. *Erasure Principal derives principal types: $\vdash_{\text{EG}} \eta\ \{C\}\ \Delta$ iff $\vdash_{\text{EP}} C : \Delta$.*

Note that Erasure Principal contains no multiple typings. Indeed, the complexity analysis of Theorem 4 carries over unchanged to Erasure Principal, showing that it is $O(nv^3)$.

5 Procedures

In this section we outline the extension of the FST system to a procedural language to obtain a context-sensitive procedural analysis. In a type-based setting a standard approach is to use type variables to represent any possible calling context in a parametric

way. This in turn requires the generation of constraints on the values of such type variables. The key observation here is that the underlying FST system is already sufficiently "polymorphic" to enable a smooth extension of the system to procedures without the need for type variables and type constraints. Because of recursion, the algorithm requires a fixed-point iteration over the analysis of procedure bodies, but this remains polynomial-time.

First let us divide the set of program variables into two disjoint sets $\{x_1, x_2, \ldots\}$ and $\{y_1, y_2 \ldots\}$ which will be used for a procedure's formal in-parameters and formal out-parameters, respectively. We assume that procedure names form a finite indexed set $\mathit{ProcName} = \{p_i\}_{i \in A}$. A program Prog is a set

$$\mathit{Prog} = \{p_i(\mathbf{in}\ x_1, \ldots, x_{n_i}; \mathbf{out}\ y_1, \ldots, y_{m_i})\ C_i\}_{i \in A}$$

where $n_i, m_i \geq 0$ and $\mathrm{fv}(C_i) \cap \mathit{PVar} \subseteq \{x_1, \ldots, x_{n_i}, y_1, \ldots, y_{m_i}\}$ (no global variables). The grammar of commands from Section 4 is extended with procedure calls:

$$C ::= \cdots \mid\ p_i(E_1, \ldots E_{n_i}; z_1, \ldots, z_{m_i})$$

where the actual out-parameters $z_1, \ldots, z_{m_i}$ are required to be distinct. For the operational semantics we can assume that there is a distinguished procedure $\mathit{main} \in \mathit{ProcName}$ defined with zero parameters. The intended semantics of a procedure call is call-by-value, return-by-value. Formal out-parameters are initialised to constant values, formal in-parameters are initialised with the values of the actual in-parameters and the actual out-parameters are assigned from the formal out-parameters at the end of the procedure call.

When extending the FST system to handle procedures it is desirable to make the type system *context-sensitive*, so that the analysis of a given call takes into account the context in which it is called. One natural way to achieve this in a compositional type-based setting is to make the analysis of procedures *polymorphic* in the security levels of their parameters. To do this, in turn, would require the introduction of type variables. However, we can bypass this step altogether. Since our principal type system is already "polymorphic" we can directly extend it to handle procedures in a context-sensitive way without making any major changes to the system (such as the introduction of type variables and an algorithm based on the solution of type constraints).

Each procedure will be typed by a function $\mathit{Var} \to \mathcal{P}(\mathit{Var})$. A program typing Ψ will be a function which gives a type to each procedure, i.e., $\Psi : \mathit{ProcName} \to \mathit{Var} \to \mathcal{P}(\mathit{Var})$. Given such a program typing we can type a command. The previous typing rules are unchanged.

The basic form of the new rule to handle procedure calls is:

$$\frac{\Delta = \Psi(p_i)}{\Psi \vdash p_i(\vec{E}; \vec{z}) : (\Delta_{\mathbf{out}_i}\ ;\ \Delta\ ;\ \Delta_{\mathbf{in}_i})[v \mapsto \{v\}]_{v \in \mathit{PVar} - \vec{z}}}$$

where:

$\Delta_{\mathbf{in}_i}$ describes the initial dependencies of the formal parameters; the formal in-parameters are initially dependent on the actual in-parameters, while the formal out-parameters are initialised to constants and so start with no dependencies, thus:

$$\Delta_{\mathbf{in}_i} = \eta[x_j \mapsto \mathrm{fv}(E_j), y_k \mapsto \{\}]_{j \in \{1 \ldots n_i\}, k \in \{1 \ldots m_i\}}$$

$\Delta_{\mathbf{out}_i}$ describes the assignment of the actual out-parameters from the formal out-parameters, thus:

$$\Delta_{\mathbf{out}_i} = \eta[z_k \mapsto \{y_k, \mathbf{pc}\}]_{k \in \{1 \ldots m_i\}}$$

If we wish to think of this in terms of instantiation of the polymorphic procedure type Δ, then $\Delta_{\mathbf{in}_i}$ is the instantiation of the in-parameter types and $\Delta_{\mathbf{out}_i}$ is the instantiation of the out-parameter types. The update in the conclusion of the rule implements the local scoping of the procedure's formal parameters; since we disallow global variables, the only effect of a procedure call on program variables in the calling context is to update the actual out-parameters, thus it acts as the identity (dependency $v \mapsto \{v\}$) on all other program variables.

As it stands, this rule does not correctly track termination flows arising from the potential for non-terminating recursions. A conservative solution would be simply to apply the update $[\mathbf{t} \mathrel{+}= \{\mathbf{pc}\}]$ to the type. A more precise typing would be obtained by using an auxiliary analysis to distinguish between those procedures which are guaranteed not to recurse infinitely (in which case the $\mathbf{t}$ update is not required) and those which may; a cheap approach would simply use the structure of the program's call graph.

In order to type the commands we need a procedure typing which is consistent with the whole program. Such a typing is described by the following rule:

$$\text{Prog}\,\frac{\forall i \in A.\, \Psi \vdash C_i : \Delta_i \quad \Psi(p_i) = \Delta_i}{\vdash \mathit{Prog} : \Psi}$$

The recursive calls from a procedure are handled no differently to any other calls, so in conventional terms the type system could be said to use polymorphic recursion.

Example Consider the swap operation in the introduction represented as a procedure:

```
swap(in x1, x2; out y1, y2) y1 := x2 ; y2 := x1
```

Using the obvious syntactic sugar for in-out parameters, the code sequence in the introduction could then be written:

```
swap(in out secret1, secret2);
swap(in out public1, public2);
```

The type for swap would be $\Delta = \eta[\mathtt{y1} \mapsto \{\mathtt{x2}, \mathbf{pc}\}, \mathtt{y2} \mapsto \{\mathtt{x1}, \mathbf{pc}\}]$. Abbreviating `secret1` as `s1` and `secret2` as `s2`, and defining $\Delta_{\mathbf{in}} = \eta[\mathtt{x1} \mapsto \{\mathtt{s1}\}, \mathtt{x2} \mapsto \{\mathtt{s2}\}]$ and $\Delta_{\mathbf{out}} = \eta[\mathtt{s1} \mapsto \{\mathtt{y1}, \mathbf{pc}\}, \mathtt{s2} \mapsto \{\mathtt{y2}, \mathbf{pc}\}]$, the typing for the first call above would thus be:

$$(\Delta_{\mathbf{out}}; \Delta; \Delta_{\mathbf{in}})[v \mapsto \{v\}]_{v \in \{\mathtt{x1},\mathtt{x2},\mathtt{y1},\mathtt{y2}\}} = \eta[\mathtt{s1} \mapsto \{\mathtt{s2}, \mathbf{pc}\}, \mathtt{s2} \mapsto \{\mathtt{s1}, \mathbf{pc}\}]$$

and the typing for the second call would be analogous.

5.1 Procedure Typing, Algorithmically

Although the Prog rule describes a valid typing it is not algorithmic. To obtain the minimal valid typing we construct an ascending chain of approximations:

$$\Psi_0 = \lambda p_i.\lambda x.\,\{\}$$
$$\Psi_{n+1} = \lambda p_i.\Delta \quad \text{where } \Psi_n \vdash C_i : \Delta$$

The complexity of typing commands is unchanged (since typing procedure call is cheap), so the cost of each iteration is $O(nv^3)$ as before. The number of iterations to reach a fixed-point is constant if the call-graph is not recursive (it is bounded by the depth of the call graph). In the presence of recursion we can bound the number of iterations at $O(v^2)$ (the height of the powerset of variables multiplied by the number of variables). In Theorem 4, v denoted the number of global variables and channels in the program. Since we no longer have global variables, v now denotes the number of channel variables plus the maximum number of parameters of any procedure.

6 Related work

To our knowledge, this is the first published work which shows how flow-sensitive multi-level security typing can be achieved in polynomial time. Our own previous work [HS06] includes an "algorithmic" type system which has exponential complexity. We also showed that the system of Amtoft and Banerjee [AB04] is equivalent to a particular instance of ours, but the published algorithmic versions of their system [AB04, AB07] are also exponential.

We are not claiming, however, that the core algorithm presented in the current paper is optimal. The specialised principal type systems we have described effectively reduce the general security typing problem to a pure dependency analysis and there are a number of previously published polynomial algorithms for implementing essentially similar dependency analyses:

- The work of Banâtre et al [BB93a, BBL94, BB93b] presents dependency analyses which are similar to the Amtoft and Banerjee system. [BBL94] in particular is for a similar language and takes an algorithmic approach. The algorithm involves constructing and then traversing a graph whose nodes correspond to program points. [BB93b] is one of the only papers which attempts formally to relate a dependency analysis to multi-level security analysis. The account is not entirely satisfactory, since the details of the multi-level analysis are not made explicit, but the conclusion is that the dependency analysis subsumes multi-level security analysis. This is also implicit in Andrews and Reitman's information flow logic [AR80], whereby a logical flow deduction is made independently of a particular policy assigning security levels to variables. The principal typings result of [HS06] confirms that conclusion but also shows that (a) dependency analysis is itself a *special case* of flow-sensitive multi-level security analysis and (b) if multi-level security in a given lattice is the property of interest, dependency analysis doesn't provide any additional precision.

– Algorithms for program slicing also incorporate dependency analysis and it is intuitively clear that they could be adapted to implement the principal type systems of the current paper. Weiser [Wei84] describes an $O(n^2)$ algorithm for building an individual slice, which would yield an $O(n^2v)$ algorithm for calculating the full matrix of dependencies (applying it once per variable).
  Later work by Horwitz et al [HRB90] casts the slicing problem as a graph traversal problem and extends the basic algorithm to the inter-procedural case, using a powerful grammar-based technique to analyse procedure calls in a context-sensitive manner. Hammer and Snelting [HS09, Ham10] explicitly apply this graph-based approach to the problem of information flow analysis. The algorithm presented by Horwitz et al is polynomial but the accompanying algorithmic analysis is stated for measures which are specific to their grammar constructions, preventing any straightforward comparison with the algorithm sketched for procedures in the current paper. More fundamentally, it is not clear how to relate the relative precision of the two approaches.
– The dependency analysis of Bergeretti and Carré [BC85] shares the same motivations as the slicing work (aiding with program comprehension, testing and debugging). Although the analysis is described in a more informal and arguably less straightforward way, their algorithm seems to be essentially equivalent to the basic principal types system described in the current paper and indeed their analysis of its complexity is $O(nv^3)$, in agreement with ours. The paper does not deal with the extension to recursive procedures. This work forms the basis of the information flow analysis of the commercial *Spark Examiner tool* [CH04], suggesting that it is algorithmically adequate.

A number of papers deal with the implementation of security type systems for similar languages e.g. [VS97, DS06] and for more complex ones [PS03], but these all have flow-insensitive treatments of imperative variables. Other algorithmic but non type-based treatments of flow-sensitive information flow include Clark et al's flow logic approach [CHH02].

Regarding erasure there is rather little prior work; the type system of the authors [HS08] and the concurrently developed system described by Chong and Myers [CM08] are perhaps the only examples. Chong and Myers do not describe an algorithm, although there approach is implemented in a restricted form as part of the Jif compiler. Their approach is incomparable because it concerns a different kind of erasure specification: it is assumed that there is an additional runtime mechanism which will overwrite all data with a certain label at a designated erasure time. The purpose of the types system is to ensure that there will be no sensitive data "left behind" when this is done. This makes less work for the static analysis and one can get away with a flow-insensitive system.

7 Conclusions and Future Work

We have presented a new approach to type-based security analysis which hinges on specialisation to principal types. The approach leads to a novel high-level structural description of a principal typing which has a direct algorithmic reading. By taking

advantage of principality we provide polynomial complexity for systems which were previously presented in an implicitly exponential manner.

One direction for future work would be to see if this development can be carried over to richer language features (e.g. [ABB06]). Do dynamic allocation, structured data and aliasing fundamentally change the algorithmic approach?

Another direction would be to consider the theoretical question of expressiveness. Among analyses which extract no information about expressions beyond the free variables that they contain, is the analysis optimal? In the context of slicing, Weiser gives an example which can be used to show that our analysis is not optimal ([Wei84], Fig 3). However if we consider the class of analyses which are also invariant under loop and recursion unrolling (as we believe ours is) then we suspect that an optimality result may be within reach.

Acknowledgements This work was partly funded by the European Commission under the WebSand project and the Swedish research agencies SSF and VR. The first author carried out part of this work while on sabbatical, very kindly hosted by Mark Harman's CREST group at UCL. Discussions with Tobias Gedell and Daniel Hedin in the early stages of this work provided valuable insights. Jens Krinke helped us navigate the related work on program slicing. Niklas Broberg and Josef Svenningsson provided helpful comments on an earlier draft. Thanks to the anonymous referees for their very helpful comments and suggestions for improvement.

References

[AB04] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *SAS 2004 (11th Static Analysis Symposium), Verona, Italy, August 2004*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.

[AB07] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming*, 64(1):3–28, 2007.

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 91–102. ACM, 2006.

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, 2008.

[AR80] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[BB93a] J.-P. Banâtre and C. Bryce. Information flow control in a parallel language framework. In *Proc. IEEE Computer Security Foundations Workshop*, pages 39–52, June 1993.

[BB93b] Jean-Pierre Banâtre and Ciaran Bryce. A security proof system for networks of communicating processes. Research Report RR-2042, INRIA, 1993.

[BBL94] J.-P. Banâtre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *Proc. European Symp. on Research in Computer Security*, volume 875 of *LNCS*, pages 55–73. Springer-Verlag, 1994.

[BC85] Jean-Francois Bergeretti and Bernard Carré. Information-flow and data-flow analysis of while-programs. *ACM TOPLAS*, 7(1):37–61, 1985.

[BC01] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*, pages 382–395. Springer-Verlag, July 2001.

[Bou05] Gérard Boudol. On typing information flow. In *ICTAC*, pages 366–380, 2005.

[CH04] Roderick Chapman and Adrian Hilton. Enforcing security and safety models with an information flow analysis tool. *Ada Lett.*, XXIV(4):39–46, 2004.

[CHH02] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *Journal of Computer Languages*, 28(1):3–28, April 2002.

[CM08] Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *CSF*, pages 98–111. IEEE Computer Society, 2008.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[DS06] Zhenyue Deng and Geoffrey Smith. Type inference and informative error reporting for secure information flow. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 543–548, New York, NY, USA, 2006. ACM.

[DS09] D. Demange and David Sands. All Secrets Great and Small. In *Programming Languages and Systems. 18th European Symposium on Programming, ESOP 2009*, number 5502 in LNCS, pages 207–221. Springer Verlag, 2009.

[Ham10] Christian Hammer. Experiences with pdg-based ifc. In *Engineering Secure Software and Systems, Second International Symposium*, pages 44–60, 2010.

[HRB90] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[HS06] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06, Proceedings of the 33rd Annual. ACM SIGPLAN - SIGACT. Symposium. on Principles of Programming Languages*, January 2006.

[HS08] Sebastian Hunt and David Sands. Just forget it - the semantics and enforcement of information erasure. In *Proc. European Symp. on Programming*, pages 239–253, 2008.

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 14–23, 1989.

[Pro59] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *IRE-AIEE-ACM '59 (Eastern): Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, New York, NY, USA, 1959. ACM.

[PS03] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, January 2003.

[Smi01] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[VS97] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, April 1997.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[Wel02] J. B. Wells. The essence of principal typings. In *Proc. International Colloquium on Automata, Languages and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.