Just Forget It
The Semantics and Enforcement of Information Erasure

Sebastian Hunt[1] and David Sands[2]

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

Abstract. There are many settings in which sensitive information is made available to a system or organisation for a specific purpose, on the understanding that it will be erased once that purpose has been fulfilled. A familiar example is that of online credit card transactions: a customer typically provides credit card details to a payment system on the understanding that the following promises are kept: (i) Noninterference (NI): the card details may flow to the bank (in order that the payment can be authorised) but not to other users of the system; (ii) Erasure: the payment system will not retain any record of the card details once the transaction is complete. This example shows that we need to reason about NI and erasure in combination, and that we need to consider interactive systems: the card details are used in the interaction between the principals, and then erased; without the interaction, the card details could be dispensed with altogether and erasure would be unnecessary. The contributions of this paper are as follows. (i) We show that an end-to-end erasure property can be encoded as a "flow sensitive" noninterference property. (ii) By a judicious choice of language construct to support erasure policies, we successfully adapt this result to an interactive setting. (iii) We use this result to design a type system which guarantees that well typed programs are properly erasing. Although erasure policies have been discussed in earlier papers, this appears to be the first static analysis to enforce erasure.

1 Information Erasure

There are many settings in which sensitive information is made available to a system or organisation for a specific purpose, on the understanding that it will be erased once that purpose has been fulfilled. Common examples involve erasure of some authentication token, such as voter identity in e-voting, or biometric data in fingerprint-activated left-luggage lockers. A more everyday example is an online credit card transaction. A customer typically provides credit card details to a payment system on the understanding that the following promises are kept:

Noninterference (NI): the card details may flow to the bank (in order that the payement can be authorised) but not to other users of the system;

Erasure: the payment system will *not* retain any record of the card details once the transaction is complete.

In this case, erasure ensures that the transaction does not make the customer or bank vulnerable to breaches of security in the payment system which occur after the transaction is complete. Two aspects of erasure are illustrated by this example:

Draft of April 29, 2008. Extended version of an article in: Proceedings of ESOP'08, 17th European Symposium on Programming, Budapest, 29 March - 6 April, 2008. Springer-Verlag (LNCS)

1. We need to be able to reason about NI and erasure in combination: we show that flow sensitive NI combined with erasure is equivalent to a re-classification of the erased input.
2. To give a satisfactory account of erasure, we need to consider *interactive* systems: the card details are used in the interaction between the customer, the payment system and the bank, and *then* erased; without the interaction, the card details could be dispensed with altogether and erasure would be unnecessary.

Background The idea and motivations for studying erasure properties of programs come from recent work of Chong and Myers [CM05], and we borrow some notation from that paper. Their paper deals with expressive temporal information flow policies for program variables which include combinations of erasure and declassification. In their simplest form, erasure policies are written in the form $a \; {}^{c}\!\nearrow \; b$, and are used to describe a variable whose security level is initially a, but which is erased to level b as soon as condition c (in principle an arbitrary property of the computation) is satisfied. Policies as described in [CM05] are quite complex (expressive), and their semantics is necessarily quite involved. It is perhaps not surprising that they have not described an enforcement mechanism (e.g. a type system) for their policy language.

In this paper we take a fresh look at the erasure problem with a much less ambitious policy language. We focus on just erasure, independently from declassification concerns. We show how, together with a judicious choice of language construct to support erasure policies, we can take advantage of the close relationship between erasure semantics and noninterference to provide, to our knowledge, the first static analysis to enforce erasure policies.

Summary We begin (Section 2) by considering what we call *end-to-end* erasure for non interactive programs. Consider the following trivial program: $y := y + 1 \; ; \; cc := 0$. This program erases (the initial value of) cc. On the other hand, $(\textbf{if} \; \mathit{isVisa}(cc) \; y := y + 1) \; ; \; cc := 0$ does not erase cc, since some information about cc is retained by y. More generally (following [CM05]) we talk about erasure of a variable *to a higher security level*. In this very simple setting we show that:

- an end-to-end erasure property can be encoded as a "flow sensitive" noninterference property (Proposition 1), and
- if we also require that the program is noninterfering, then this is a necessary and sufficient condition for erasure (Proposition 2).

End-to-end erasure is too simple to be useful in itself. In Section 3 we move on to the study of erasure in the presence of fresh inputs and program outputs. Consider for example the program to the right. Here the erasure property we might want is that no information about the input cc in the first line of the loop body can be observed after the transaction (the loop body) is complete. In this case the input is *not* erased because it is still present in $\mathit{payment}$, so if the

```
$\textbf{while}\ \mathit{serverUp}\ \{$
  $\textbf{input}\ \mathit{cc}\ \textbf{from}\ \mathit{user}$
  $\textbf{input}\ \mathit{details}\ \textbf{from}\ \mathit{user}$
  $\mathit{payment} := \mathit{process}(cc)$
  $\textbf{output}\ \mathit{payment}\ \textbf{to}\ \mathit{bank}$
  $\mathit{custInfo} := \mathit{custInfo} \oplus \mathit{details}$
  $cc := 0$
$\} \ldots$
```

server goes down the credit card information of the last transaction could be retrieved from this variable and output by the system.

Defining what it means for a program to erase data in the general case is potentially complex and, we suspect, correspondingly difficult to enforce. The key idea that we introduce in Section 3 is a simple language mechanism to specify a well behaved class of erasure policies. We introduce a block structured input command of the form $\mathbf{input}\ x\ \mathbf{from}\ a\ \mathbf{erased\ in}\ C$ (the exact syntactic form accommodates a more general notion than this and is written $\mathbf{input}\ \ x\ :\ a \nearrow b\ \ \mathbf{in}\ \ C$) thereby tying the semantic lifetime of the input (from the point of view of certain observers) to code block C. This facilitates the subsequent development as follows:

- the definition of when a program correctly enforces such erasure policies (we call such a program *input erasing*) becomes easy to state (Definition 4)
- because of the block structured nature of the erasure policy, we can apply ideas from Section 2 to determine a local end-to-end style erasure condition (Definition 6) which, as for end-to-end erasure, can also be expressed as a reclassified noninterference property (Theorem 1)
- we can then show that the local erasure condition together with a suitable noninterference property is sufficient to guarantee that a program is input erasing (Theorem 2).

Our final contribution (Section 4) is to use this local characterisation of erasure to design a type system which guarantees that well typed programs are input erasing. The type system is a direct adaptation (extension) of a flow sensitive type system for noninterference described in [HS06].

Section 5 discusses some of the subtleties of erasure and the computation model. Section 6 concludes, revisiting related work and sketching some ideas for further work.

2 End-to-End Erasure

We start by considering erasure in its "purest" form. Consider programs which just transform some initial memory state to a final memory state. Concretely, we can consider a simple *while* language with no input or output commands (essentially the language described in Figure 2 with all the input-output machinery removed). The semantics of this language can be given as a small-step deterministic transition relation on configurations, where terminating computations have the form $\langle C, s\rangle \twoheadrightarrow \langle \mathbf{skip}, t\rangle$ (here C is a program and s, t are memory *states*: finite mappings from the set Var of variable names to values).

2.1 Flow sensitive End-to-End Noninterference

As in [HS06] we consider a flow sensitive form of noninterference. Let Γ, Γ' be finite mappings from variable names to elements of $\langle \mathcal{L}, \sqsubseteq, \sqcup, \sqcap\rangle$ a lattice of security levels. We will call these *security type assignments*. We write $s =_X t$ to mean that states s and t agree on all variables in the set X. For $a \in \mathcal{L}$ we write $\Gamma \vdash s =_a t$ to mean that s and t are equal to all observers at or below security level a, with respect to the security type assignment Γ. That is: $\Gamma \vdash s =_a t$ iff $s =_X t$ where $X = \{x | \Gamma(x) \sqsubseteq a\}$.

Definition 1 (Noninterference (NI)). *A command C is noninterfering from Γ to Γ', written $\Gamma\ \{C\}\ \Gamma'$, iff, for all $a \in \mathcal{L}$, if $\Gamma \vdash s =_a t$ and $\langle C, s\rangle \twoheadrightarrow \langle \mathbf{skip}, s'\rangle$ then $\langle C, t\rangle \twoheadrightarrow \langle \mathbf{skip}, t'\rangle$ for some t' such that $\Gamma' \vdash s' =_a t'$.*

(Note that, since programs are deterministic, if t' exists - ie if the program terminates - it is unique.) In other words, noninterference says that if two initial states are indistinguishable to an observer at a (with respect to Γ) then the resulting states will also be indistinguishable (with respect to Γ'). Note that, unlike [HS06], this is a termination *sensitive* NI property, meaning that we do *not* allow information leaks through termination/nontermination behaviour. We chose this stronger variant because it is better suited to a computational model with input-output (Section 3).

2.2 End-to-End Erasure

In what follows we have chosen to model erasure of the information stored in individual variables. Our choice is essentially pragmatic: it allows us to express the key ideas in a simple way while supporting reasonably expressive erasure policies. Other choices are possible. For example we could model erasure of all information stored at a given security level, or, conversely, partial erasure of the information stored in a variable. To be more general still, one could model erasure of arbitrary projections on the program state – and such things could be done in the PER model [SS01] or using abstract noninterference [GM04]).

We define end-to-end erasure as a simple information flow property. In its simplest form, say that a program *completely erases* the information in variable x if varying (just) the information in x prior to execution has no effect on the final program state. In fact we want to be more general than this (following [CM05]). We will say that x is erased *to some level* b, if varying x leaves the final state unchanged from the viewpoint of all observers except those at level b or above. In what follows we write $\neg x$ for $Var - \{x\}$.

Definition 2 (End-to-End Erasure). *Command C erases x to b in Γ', written $C : x \nearrow b$ in Γ', iff, whenever $s =_{\neg x} t$ and $\langle C, s\rangle \twoheadrightarrow \langle \mathbf{skip}, s'\rangle$ then $\langle C, t\rangle \twoheadrightarrow \langle \mathbf{skip}, t'\rangle$, for some t' such that $\forall c \not\sqsupseteq b$, $\Gamma' \vdash s' =_c t'$.*

Note that we can recover complete erasure from the more general definition, in the form $C : x \nearrow \top$ in Γ, as long as we have some security level $\top$ such that, for all variables y, $\Gamma(y) \not\sqsupseteq \top$.

Consider the example programs in Figure 1. We have $P_1 : z_L \nearrow H$ in Γ, but P_2 does *not* erase $z_L \nearrow H$ because although z_L itself is physically overwritten, information about the initial value of z_L is still present in y_M. The same goes for P_3: it does not erase z_L to H, this time because of an indirect information flow to y_M.

Typically, we will wish to enforce policies in which erasure is required *in addition* to NI. The programs in Figure 1 satisfy $\Gamma\ \{P_i\}\ \Gamma$ $(i = 1, 2, 3)$. If we replaced $z_L := 0$ with $z_L := y_M$ in P_1 the program would still erase z_L to H, but would not be noninterfering from Γ to Γ.

2.3 Relating End-to-End Erasure and NI

It is clear from the definitions that end-to-end erasure and noninterference are closely related. In later sections we exploit this relationship in both the design of an erasure

$$\begin{array}{lll}
P_1 : x_H := x_H + y_M + z_L & P_2 : x_H := x_H + y_M + z_L & P_3 : x_H := x_H + y_M + z_L \\
\, y_M := y_M + 2 & \, y_M := y_M + z_L & \, \textbf{if } (z_L = 0)\ y_M := y_M + 1 \\
\, z_L := 0 & \, z_L := 0 & \, z_L := 0
\end{array}$$

Fig. 1. Example programs, assuming security type assignment $\Gamma = [x_H {\mapsto} H, y_M {\mapsto} M, z_L {\mapsto} L]$ with respect to the three point lattice $L \sqsubseteq M \sqsubseteq H$.

policy mechanism, and in the adaptation of the flow sensitive type system from [HS06] to produce a type system which also enforces erasure policies. The key observation is that every erasure property can be enforced by requiring a related NI property.

Proposition 1. *If $\Gamma[x \mapsto b]\ \{C\}\ \Gamma'$ then $C : x \nearrow b$ in Γ'.*

Proof. Assume lhs. Suppose $s =_{\neg x} t$ and $c \not\sqsupseteq b$. From the definitions and by assumption of lhs, it suffices to show that $\Gamma[x \mapsto b] \vdash s =_c t$: this is immediate from $s =_{\neg x} t$ and $\Gamma[x \mapsto b](x) = b \not\sqsubseteq c$. □

For example, the Proposition tells us that we can verify $P_1 : z_L \nearrow H$ (Figure 1) by showing that $\Gamma[x_L {\mapsto} H]\ \{P_1\}\ \Gamma$, and this can be done, for example, using the type system from [HS06].

While useful, this leaves open the possibility that the reclassified NI condition of Proposition 1 is too strong in general, requiring much more than is necessary to ensure erasure. In practice, however, we wish to enforce erasure *and* noninterference together. The following result shows that, if we already require the NI property $\Gamma\ \{C\}\ \Gamma'$, then the reclassified NI property $\Gamma[x \mapsto b]\ \{C\}\ \Gamma'$ is *precisely* what we need to ensure that x is erased to b.

Proposition 2. *If $\Gamma\ \{C\}\ \Gamma'$ then $C : x \nearrow b$ in $\Gamma' \iff \Gamma[x \mapsto b]\ \{C\}\ \Gamma'$.*

Proof. Subsumed by Theorem 1.

3 Erasure in the Presence of Input-Output

The previous section showed how end-to-end erasure policies can be determined by using reclassification and noninterference. But end-to-end erasure is not the kind of policy we ultimately want to enforce. If all the attacker does is literally observe the final values of a computation then Proposition 2 really tells us that an erasure policy is just a way to fix a noninterference policy for which some data was assigned a level which is too low.

Our task now is to generalise the notion of erasure to make it more meaningful and more expressive. To do this we consider a system with inputs and outputs, and a notion of erasure at an intermediate program point. For simplicity, we will identify security levels with channels, thus for each $a \in \mathcal{L}$, we assume exactly one channel, also named a, which carries data at level a (c.f. [OCC06]).

It is tempting (and potentially expressive) to introduce separate constructs for input

$$\begin{array}{l}\textbf{input}\ x\ \textbf{from}\ a\\ \textbf{if}\ (x=0)(x:=-1;\ \textbf{erased}\ x\ \textbf{to}\ b)\\ \textbf{output}\ x\ \textbf{on}\ a\end{array}$$

and erasure. But consider the example to the right. Clearly, x is literally overwritten with a constant in every run which passes the erasure assertion. Intuitively though, this program should be rejected, since an observer of outputs on a can still deduce something about the erased data. This is an example of one particular problem; there are potentially many such problems compounded by the interaction between different erasure operations and the deductions an observer can make though inputs and outputs.

Our key idea is to avoid these problems by combining input and erasure into a single block structured command:

$$\textbf{input}\ x : a \nearrow b\ \textbf{in}\ C$$

which can be read as the policy "input x on channel a then compute C, after which x will have been erased to level b". By associating the lifetime of the data with the erasure policy in a block-structured way we avoid some of the subtle problems of indirect information flow interacting with the erasure policy. More importantly, we will show that we can apply the end-to-end erasure definition locally to the command C to achieve a meaningful global erasure.

To show that this is really the case we must first extend our definitions of noninterference and erasure to take into account the fact that the language now has IO.

3.1 A Language with Input and Output

To be concrete let us take the simple *while* language and add input as an erasure declaration as above, and a simple output statement. For the operational semantics of this language we assume the existence of an infinite input stream for each security level. We let I denote the set of input streams and, for any level a, I_a denotes the stream of a-inputs, and $I_a(m)$, $m > 0$ denotes the m^{th} input on channel a.

We assume a small-step operational semantics with configurations of the form $\langle C, s, \mathbf{i}\rangle$, where C and s are as before and $\mathbf{i} \in \mathcal{L} \to \mathbb{N}$ is the input stream pointer which records how much of the input streams have been consumed so far.

Transitions are written in the form $I \vdash \langle C, s, \mathbf{i}\rangle \xrightarrow{\ell} \langle C', t, \mathbf{i}'\rangle$ where the label ℓ is either an input event $a?v$, a silent transition τ, or an output event $a!v$. We will often omit the label τ. The syntax and semantics are given in Figure 2. The input streams I are left implicit in the rules. We assume an expression evaluator $[\![E]\!]s$ which produces a value from an expression and an environment. We implicitly assume well-typedness for expressions.

A "vanilla" input command $\textbf{input}\ x\ \textbf{from}\ a$, i.e. one which is not associated with an erasure property, can be defined as a shorthand for the trivial erasure $\textbf{input}\ x : a \nearrow a\ \textbf{in}\ \textbf{skip}$ (it is trivially erasing because "after executing $\textbf{skip}$ the value input on channel a will only be visible at level a or above").

From the single step evaluation relation we define the zero-or-more-step relation $\overset{\alpha}{\twoheadrightarrow}$, labelled with a sequence of non-silent events, in the obvious way. We write $c_1 \twoheadrightarrow c_2$ to mean that $c_1 \overset{\alpha}{\twoheadrightarrow} c_2$ for some (possibly empty) α and $c_1 \overset{\alpha}{\twoheadrightarrow}$ to mean $\exists c_2 . c_1 \overset{\alpha}{\twoheadrightarrow} c_2$.

$$\begin{array}{rl} \text{Expressions} & E ::= n \mid x \mid E \; op \; E' \\ \text{Commands} & C ::= \mathbf{skip} \mid x := E \mid C_1 \; ; \; C_2 \mid \mathbf{if} \; E \; C_1 \; C_2 \mid \mathbf{while} \; E \; C \\ & \qquad \mid \mathbf{input} \; x : a \nearrow b \; \mathbf{in} \; C \mid \mathbf{output} \; E \; \mathbf{on} \; a \\ \text{Reduction Contexts} & R ::= [\cdot] \mid (R[\cdot] \; ; \; C) \end{array}$$

$$\frac{I_a(n) = v \qquad n = \mathbf{i}(a) + 1}{\langle \mathbf{input} \; x : a \nearrow b \; \mathbf{in} \; C, s, \mathbf{i} \rangle \xrightarrow{a?v} \langle C, s[x \mapsto v], \mathbf{i}[a \mapsto n] \rangle}$$

$$\frac{[\![E]\!]s = v}{\langle x := E, s, \mathbf{i} \rangle \xrightarrow{\tau} \langle \mathbf{skip}, s[x \mapsto v], \mathbf{i} \rangle} \qquad \frac{[\![E]\!]s = v}{\langle \mathbf{output} \; E \; \mathbf{on} \; a, s, \mathbf{i} \rangle \xrightarrow{a!v} \langle \mathbf{skip}, s, \mathbf{i} \rangle}$$

$$\frac{[\![E]\!]s = v \in \{\mathbf{true}, \mathbf{false}\}}{\langle \mathbf{if} \; E \; C_{\mathbf{true}} \; C_{\mathbf{false}}, s, \mathbf{i} \rangle \xrightarrow{\tau} \langle C_v, s, \mathbf{i} \rangle}$$

$$\frac{}{\langle \mathbf{while} \; E \; C, s, \mathbf{i} \rangle \xrightarrow{\tau} \langle \mathbf{if} \; E \; (C \; ; \; \mathbf{while} \; E \; C) \; \mathbf{skip}, s, \mathbf{i} \rangle}$$

$$\frac{}{\langle (\mathbf{skip} \; ; \; C), s, \mathbf{i} \rangle \xrightarrow{\tau} \langle C, s, \mathbf{i} \rangle} \qquad \frac{\langle C, s, \mathbf{i} \rangle \xrightarrow{\ell} \langle C', s', \mathbf{i}' \rangle}{\langle R[C], s, \mathbf{i} \rangle \xrightarrow{\ell} \langle R[C'], s', \mathbf{i}' \rangle}$$

Fig. 2. Syntax and Semantics

3.2 Noninterference and Input Erasure

We extend the equality relation $=_a$ to input streams (and input stream pointers) by saying $I =_a I'$ ($\mathbf{i} =_a \mathbf{j}$) whenever $I_c = I'_c$ ($\mathbf{i}(c) = \mathbf{j}(c)$) for all $c \sqsubseteq a$. We write $\alpha =_a \beta$ to mean equality of the projections of α and β to all labels on channel a or lower.

Definition 3 (Input-Output Noninterference). *We define a command C to be* input-output noninterfering *if for all $a \in \mathcal{L}$, and all input streams I and I', if $I =_a I'$ and $I \vdash \langle C, s, \mathbf{i} \rangle \overset{\alpha}{\twoheadrightarrow}$ then $I \vdash \langle C, s, \mathbf{i} \rangle \overset{\beta}{\twoheadrightarrow}$ for some β such that $\alpha =_a \beta$.*

Let us now turn to the definition of the erasure property that we want. It says that in any execution, once control has reached the end of the input block $\mathbf{input} \; x : a \nearrow b \; \mathbf{in} \; C$ – i.e. once we have finished executing C – then no information about x should be visible through subsequent input or output events except at level b or higher.

Definition 4 (Input Erasure). *We say that a command C_0 is* input erasing *if for all input streams I the following property holds. Suppose we have a computation of the following form:*

$$I \vdash \langle C_0, s_0, \mathbf{i}_0 \rangle \twoheadrightarrow \langle R[\mathbf{input} \; x : a \nearrow b \; \mathbf{in} \; C], s, \mathbf{i} \rangle \twoheadrightarrow \langle R[\mathbf{skip}], s_1, \mathbf{i}_1 \rangle \overset{\alpha}{\twoheadrightarrow}$$

where the computation $R[\mathbf{input} \; x : a \nearrow b \; \mathbf{in} \; C] \twoheadrightarrow R[\mathbf{skip}]$ is independent of $R[\cdot]$. Let I' be an input stream which only differs from I on channel a at input position $\mathbf{i}(a) + 1$. Then the input erasing condition requires that there exists a computation of the following form:

$$I' \vdash \langle C_0, s_0, \mathbf{i}_0 \rangle \twoheadrightarrow \langle R[\mathbf{input} \; x : a \nearrow b \; \mathbf{in} \; C], s, \mathbf{i} \rangle \twoheadrightarrow \langle R[\mathbf{skip}], t_1, \mathbf{j}_1 \rangle \overset{\beta}{\twoheadrightarrow}$$

such that $\forall c \not\sqsupseteq b$ *we have* $s_i =_c t_i$, $\mathbf{i}_i =_c \mathbf{j}_i$ *(*$i = 1, 2$*) and* $\alpha =_c \beta$.

Note that the requirement that $I' \vdash \langle C_0, s_0, \mathbf{i}_0\rangle \twoheadrightarrow \langle R[\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C], s, \mathbf{i}\rangle$ is actually vacuous since the computation has not yet reached the point at which I and I' differ. The start state s_0 and $\mathbf{i}_0$ in the above are universally quantified, but could be fixed. A natural choice for an initial input pointer would of course be $\lambda a.0$.

The following proposition formalises the sense in which the "vanilla" input is always erasing:

Proposition 3. *If* C *is input-output noninterfering and if each input command in* C *has the form* $\mathbf{input}\ x : a \nearrow a\ \mathbf{in}\ \mathbf{skip}$ *for some* x *and* a *then* C *is input erasing.*

Proof. Let C be ain input-output noninterferring command with input commands restricted as above. Let $\mathbb{C} = \langle C, s_0, \mathbf{i}_0\rangle$. Any computation involving an input command with an input stream I has the form:

$$I \vdash \mathbb{C} \overset{\alpha_0}{\twoheadrightarrow} \langle R[\mathbf{input}\ x : a \nearrow a\ \mathbf{in}\ \mathbf{skip}], s, \mathbf{i}\rangle \overset{a?v}{\rightarrow} \langle R[\mathbf{skip}], s[x \mapsto v], \mathbf{i}_1\rangle \overset{\alpha}{\twoheadrightarrow}$$

Suppose I' differs only from I at the position corresponding to the input v above. Then we have a computation of the form:

$$I' \vdash \mathbb{C} \overset{\alpha_0}{\twoheadrightarrow} \langle R[\mathbf{input}\ x : a \nearrow a\ \mathbf{in}\ \mathbf{skip}], s, \mathbf{i}\rangle \overset{a?w}{\rightarrow} \langle R[\mathbf{skip}], s[x \mapsto w], \mathbf{i}_1\rangle$$

Since C is assumed to be input noninterfering then if we take any $c \not\sqsupseteq a$ we have $I =_c I'$, and hence $I' \vdash \mathbb{C} \overset{\beta_0}{\twoheadrightarrow}$ for some β_0 such that $\alpha_0(a?v)\alpha =_c \beta_0$. Since the language is deterministic, it follows that β_0 must have the form $\alpha_0(a?w)\beta$ for some β such that $\alpha =_c \beta$, and where $\langle R[\mathbf{skip}], s[x \mapsto w], \mathbf{i}_1\rangle \overset{\beta}{\twoheadrightarrow}$ as required. □

3.3 Characterising Input Erasure with a Local Erasure Condition

In this section we develop a local characterisation of erasure – a generalisation of end-to-end erasure which we can apply locally to the command $\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C$ – which will help us establish the "global" input erasure condition.

To do this we will need to work with a stronger notion of noninterference than input-output noninterference. Although the definition of input-output noninterference is a reasonable top level definition (for more discussion on this point see section 5) it is difficult to work with since it says nothing about the state. For example it is *not* compositional with respect to sequential composition: $C_1 = \mathbf{input}\ x\ \mathbf{on}\ H\ ;\ \mathbf{if}\ x\ \mathbf{then}\ y := 1$ is IO-noninterfering, and so is $C_2 = \mathbf{output}\ y\ \mathbf{on}\ L$, but $C_1 ; C_2$ is not. It is convenient therefore to work with a stronger definition which also looks at the initial and terminal state (in the case that the program terminates).

Definition 5 (Stateful Input-Output Noninterference). *A command* C *is noninterfering from* Γ *to* Γ'*, written* $\Gamma\ \{C\}\ \Gamma'$*, iff, for all* $a \in \mathcal{L}$*, and all input streams* I*,* I'*, if* $\Gamma \vdash s =_a t$*,* $I =_a I'$*,* $\mathbf{i} =_a \mathbf{j}$ *then*

1. *if* $I \vdash \langle C, s, \mathbf{i}\rangle \overset{\alpha}{\twoheadrightarrow}$ *then* $I' \vdash \langle C, t, \mathbf{j}\rangle \overset{\beta}{\twoheadrightarrow}$ *for some* β *such that* $\alpha =_a \beta$*, and*

2. *if $I \vdash \langle C, s, \mathbf{i}\rangle \twoheadrightarrow \langle \mathbf{skip}, s', \mathbf{i}'\rangle$ then $I' \vdash \langle C, t, \mathbf{j}\rangle \twoheadrightarrow \langle \mathbf{skip}, t', \mathbf{j}'\rangle$ such that $\mathbf{i}' =_a \mathbf{j}'$ and $\Gamma' \vdash s' =_a t'$.*

Now we will define an extension of the end-to-end erasure property. The idea is that, when enforced locally on the erasing input command, the property will be sufficient to ensure the global erasure property.

The definition ensures that if a specific variable x is erased from a to b then it is neither "visible" in the state except at or above b (precisely as before) *nor* via the input pointer:

Definition 6 (Local Erasure). *Command C erases x to b in Γ', written $C : x \nearrow b$ in Γ', iff, whenever $s =_{\neg x} t$ and $I \vdash \langle C, s, \mathbf{i}_0\rangle \twoheadrightarrow \langle \mathbf{skip}, s', \mathbf{i}\rangle$ then $I \vdash \langle C, t, \mathbf{i}_0\rangle \twoheadrightarrow \langle \mathbf{skip}, t', \mathbf{j}\rangle$, for some t' and $\mathbf{j}$ such that $\forall c \not\sqsupseteq b$, $\Gamma' \vdash s' =_c t'$ and $\mathbf{i} =_c \mathbf{j}$.*

Note that we have overloaded some definitions defined in Section 2. It is reasonable to do this because they are conservative extensions of the earlier definitions. Stated more precisely:

Proposition 4. *For any command C free from input-output statements, define the IO-free semantics of C to be $\langle C, s\rangle \to \langle C', t\rangle$ iff $I \vdash \langle C, s, \mathbf{i}\rangle \xrightarrow{\tau} \langle C', t, \mathbf{i}\rangle$. For any such C we have that*

1. *The statement that C is noninterfering from Γ to Γ' is identical for Definition 1 and Definition 5.*
2. *The statement that C erases x to b in Γ' is identical for Definition 2 and Definition 6.*

Proof. Observing that input-output-free commands compute independently from the input streams, and to not modify the input stream pointers, then the result is immediate by specialisising definitions 5 and Definition 6. □

The local erasure condition ignores the input and outputs that take place before the computation is complete, but the condition nevertheless demands that $\mathbf{i} =_c \mathbf{j}$. This is motivated by the fact that the state of the input pointer can be used as a covert store to save information about the erased secret. Consider the command C defined as

$$\begin{array}{ll} \mathbf{if}\ (x \neq 0)\ (\mathbf{input}\ y\ \mathbf{on}\ M); & \\ x := 0; y := 0 & \end{array} \qquad \text{(where } L \sqsubseteq M \sqsubseteq H\text{)}$$

If we ignored the final value of the input pointers, then this command would be considered to erase x. This would be too weak for our purposes because after the erasure, information about x *will* be known to an observer at level M. To see this, consider using the command (C) in the program to the right. So for example if the M input stream has the value $0, 1 \ldots$ then the value of y output on M will be 0 if x was 0 and 1 otherwise.

$y := 0$;
$\mathbf{input}\ x : L \nearrow H\ \mathbf{in}\ C$;
$\mathbf{input}\ y\ \mathbf{on}\ M$;
$\mathbf{output}\ y\ \mathbf{on}\ M$

Reclassification In the manner of Proposition 1, we will show that the local erasure property can be characterised in terms of noninterference. But since noninterference cares about the input output events that occur during a computation, and local erasure does not, we need a way to "turn a blind eye" to input output events. Towards this end it is useful – for specification purposes only – to introduce a language construct which "hides" inputs and outputs:

Definition 7. *We extend the language with commands of the form $\widehat{C}$ with semantics*

$$\frac{\langle C, s, \mathbf{i}\rangle \xrightarrow{\alpha} \langle C', s', \mathbf{i}'\rangle}{\langle \widehat{C}, s, \mathbf{i}\rangle \xrightarrow{\tau} \langle \widehat{C'}, s', \mathbf{i}'\rangle} \qquad \frac{}{\langle \widehat{\mathbf{skip}}, s, \mathbf{i}\rangle \xrightarrow{\tau} \langle \mathbf{skip}, s, \mathbf{i}\rangle}$$

This is essentially just like the hiding operation of CSP, and is commonly used in process calculi to specify noninterference properties (see e.g. [Ros95,FG95]), except that here we are hiding *all* events, so $\widehat{C}$ behaves like C but with every input or output label of C replaced by the silent action τ.

Theorem 1 (Local Erasure as Reclassification). *If $\mathit{\Gamma}\ \{C\}\ \mathit{\Gamma}'$ then*

$$C : x \nearrow b \textit{ in } \mathit{\Gamma}' \iff \mathit{\Gamma}[x \mapsto b]\ \{\widehat{C}\}\ \mathit{\Gamma}'$$

The theorem says that to check noninterference and erasure for a command it is necessary and sufficient to check noninterference and a reclassified noninterference property but where input and output labels are ignored.

Proof. First we note that we can easily specialise the definition of noninterference to the case where programs perform no visible IO. In this way we can see that $\mathit{\Gamma}[x \mapsto b]\ \{\widehat{C}\}\ \mathit{\Gamma}'$ if and only if for all a,

$$\begin{aligned} &I =_a I' \ \wedge\ \mathit{\Gamma}[x \mapsto b] \vdash s =_a t \ \wedge\ \mathbf{i} =_a \mathbf{j} \ \wedge\ I \vdash \langle \widehat{C}, s, \mathbf{i}\rangle \twoheadrightarrow \langle \mathbf{skip}, s', \mathbf{i}'\rangle \\ &\qquad \Rightarrow \exists t', \mathbf{j}'.\quad \mathit{\Gamma}' \vdash s' =_a t' \ \wedge\ \mathbf{i}' =_a \mathbf{j}' \ \wedge\ I' \vdash \langle \widehat{C}, t, \mathbf{j}\rangle \twoheadrightarrow \langle \mathbf{skip}, t', \mathbf{j}'\rangle \end{aligned}$$

The following properties are easy consequences of the definitions and semantics of hiding, and will be used freely in what follows: (i) $\mathit{\Gamma}\ \{C\}\ \mathit{\Gamma}' \Rightarrow \mathit{\Gamma}\ \{\widehat{C}\}\ \mathit{\Gamma}'$ and (ii) $C : x \nearrow b$ in $\mathit{\Gamma}' \iff \widehat{C} : x \nearrow b$ in $\mathit{\Gamma}'$.

For the ($\Leftarrow$)-direction of the theorem it is sufficient to prove that $\mathit{\Gamma}[x \mapsto b]\ \{\widehat{C}\}\ \mathit{\Gamma}'$ implies $\widehat{C} : x \nearrow b$ in $\mathit{\Gamma}'$. Assume the rhs, that $s =_{\neg x} t$, and $I \vdash \langle \widehat{C}, s, \mathbf{i}\rangle \twoheadrightarrow \langle \mathbf{skip}, s', \mathbf{i}'\rangle$. Since s and t only differ at x, and in $\mathit{\Gamma}[x \mapsto b]$ we have that x is not visible at level $c \not\sqsupseteq b$ we have $\forall c \not\sqsupseteq b. \mathit{\Gamma}[x \mapsto b] \vdash s =_c t$. From the specialised noninterference property above we have that $\exists t', \mathbf{j}'.\quad \mathit{\Gamma}' \vdash s' =_c t' \wedge \mathbf{i}' =_c \mathbf{j}' \wedge I \vdash \langle \widehat{C}, t, \mathbf{j}\rangle \twoheadrightarrow \langle \mathbf{skip}, t', \mathbf{j}'\rangle$.

For the ($\Rightarrow$)-direction of the theorem, assume the premise and that $I =_a I' \wedge \mathit{\Gamma}[x \mapsto b] \vdash s =_a t \wedge \mathbf{i} =_a \mathbf{j} \wedge I \vdash \langle \widehat{C}, s, \mathbf{i}\rangle \twoheadrightarrow \langle \mathbf{skip}, s', \mathbf{i}'\rangle$.

If $a \sqsupseteq b$, consider whether or not $\mathit{\Gamma}(x) \sqsubseteq a$. In either case, $\{y | \mathit{\Gamma}(y) \sqsubseteq a\} \subseteq \{y | \mathit{\Gamma}[x \mapsto b](y) \sqsubseteq a\}$. Hence, when $a \sqsupseteq b$, $\mathit{\Gamma}[x \mapsto b] \vdash s =_a t$ implies $\mathit{\Gamma} \vdash s =_a t$ and the required $t', \mathbf{j}'$ exist by assumption of noninterference.

Assume then that $a \not\sqsupseteq b$. Let $s_1 = s[x \mapsto t(x)]$. Then we have $s =_{\neg x} s_1$, and $\mathit{\Gamma}[x \mapsto b] \vdash s_1 =_a t$. Since s_1 and t agree on x then we also have $\mathit{\Gamma} \vdash s_1 =_a t$. Since $C : x \nearrow b$ in $\mathit{\Gamma}'$ we have that $I \vdash \langle \widehat{C}, s_1, \mathbf{i} \rangle \twoheadrightarrow \langle \mathbf{skip}, s_1', \mathbf{i}_1 \rangle$ where $\forall c \not\sqsupseteq b . \mathbf{i}' =_c \mathbf{i}_1 \wedge \mathit{\Gamma}' \vdash s' =_c s_1'$. Since in particular $a \not\sqsupseteq b$ then $\mathbf{i}' =_a \mathbf{i}_1 \wedge \mathit{\Gamma}' \vdash s' =_a s_1'$. From $\mathit{\Gamma}\ \{C\}\ \mathit{\Gamma}'$ we have $\mathit{\Gamma}\ \{\widehat{C}\}\ \mathit{\Gamma}'$, and since $\mathit{\Gamma} \vdash s_1 =_a t$, we get $I' \vdash \langle \widehat{C}, t, \mathbf{j} \rangle \twoheadrightarrow \langle \mathbf{skip}, t', \mathbf{j}' \rangle$ where $\mathbf{i}_1' =_a \mathbf{j}$ and $\mathit{\Gamma}' \vdash s_1' =_a t'$. From transitivity of $=_a$ we conclude that $\mathbf{i} =_a \mathbf{j}$ and $\mathit{\Gamma}' \vdash s' =_a t'$ as required. □

3.4 From Local to Global Erasure

We have defined a local erasure condition for commands with IO. The purpose of the local condition is to provide sufficient conditions for input erasure. But in order to complete this picture we need some noninterference conditions: the local erasure property can only give input erasure if the rest of the program does not allow the erased information to flow back down to a lower level, i.e. it must have a noninterference property.

Annotations To state the noninterference assumptions we need, we will use program annotations. Annotations will provide the link to compositional program analyses such as type systems. An annotation here is just a security type assignment. The operational semantics of an annotation is transparent (otherwise it would not be an annotation!): we extend the grammar of reduction contexts with the annotated context $(R[\cdot])^{\mathit{\Gamma}}$, and specify the rule $\langle \mathbf{skip}^{\mathit{\Gamma}}, s, \mathbf{i} \rangle \rightarrow \langle \mathbf{skip}, s, \mathbf{i} \rangle$. In an annotated subterm $C^{\mathit{\Gamma}}$, the annotation $\mathit{\Gamma}$ is intended to describe the security levels of the state at the point in execution after C has been evaluated. This intuition is made concrete in the following definition which connects annotations to the noninterference property.

Definition 8 (Well-annotated Commands). *Command C_0 is* well annotated *iff:*

1. *every annotated input command $(\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C)^{\mathit{\Gamma}}$ in C_0 has the local erasure property $C : x \nearrow b$ in $\mathit{\Gamma}$;*
2. *whenever a command of the form $R[\mathbf{skip}^{\mathit{\Gamma}}]$ is reached from any computation beginning with C_0, then $\mathit{\Gamma}\ \{R[\mathbf{skip}]\}\ \mathit{\Gamma}'$ for some $\mathit{\Gamma}'$.*

Theorem 2. *If C_0 is a well-annotated command such that every input command in C_0 is annotated, then C_0 is input erasing.*

Proof. Suppose that C_0 satisfies the premise of the claim, and that

$$\begin{aligned} I \vdash \langle C_0, s_0, \mathbf{i}_0 \rangle &\twoheadrightarrow \langle R[\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C], s, \mathbf{i} \rangle \\ &\twoheadrightarrow \langle R[\mathbf{skip}], s_1, \mathbf{i}_1 \rangle \qquad\qquad \text{(independent of } R[\cdot]\text{).} \\ &\overset{\alpha}{\twoheadrightarrow} \end{aligned}$$

Since each input command is annotated we know that $R[\cdot] = R'[\cdot^{\mathit{\Gamma}}]$ for some $R'[\cdot]$ and $\mathit{\Gamma}$. Suppose further that $\mathbf{i}(a) = k$ and that $I_a(k+1) = v$. Given this, we know that the above computation has the form

$$\begin{aligned} I \vdash \langle C_0, s_0, \mathbf{i}_0 \rangle &\twoheadrightarrow \langle R'[(\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C)^{\mathit{\Gamma}}], s, \mathbf{i} \rangle \\ &\xrightarrow{a?v} \langle R'[C^{\mathit{\Gamma}}], s[x \mapsto v], \mathbf{i}[a \mapsto k+1] \rangle \\ &\twoheadrightarrow \langle R'[\mathbf{skip}^{\mathit{\Gamma}}], s_1, \mathbf{i}_1 \rangle \overset{\alpha}{\twoheadrightarrow} \end{aligned}$$

Suppose I' is like I except that $I'_a(k+1) = v'$. Now we have that

$$I' \vdash \langle C_0, s_0, \mathbf{i}_0 \rangle$$

$$\twoheadrightarrow \langle R'[(\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C)^{\Gamma}], s, \mathbf{i} \rangle \tag{i}$$

$$\xrightarrow{a?v} \langle R'[C^{\Gamma}], s[x \mapsto v'], \mathbf{i}[a \mapsto k+1] \rangle \tag{ii}$$

$$\twoheadrightarrow \langle R'[\mathbf{skip}^{\Gamma}], t_1, \mathbf{j}_1 \rangle \rightarrow \langle R'[\mathbf{skip}], t_1, \mathbf{j}_1 \rangle \tag{iii}$$

$$\overset{\beta}{\twoheadrightarrow} \tag{iv}$$

Step (i) holds because the input on which I and I' differ has not yet been reached; (ii) follows from the semantics of input. From the operational semantics it is easily seen that the command $(\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C)^{\Gamma}$ must have been present in the original program: since C_0 is well annotated we have that $C : x \nearrow b$ in Γ, hence computation (iii) exists and $\forall c \not\sqsupseteq b$, $\Gamma \vdash s_1 =_c t_1$ and $\mathbf{i}_i =_c \mathbf{j}_i$. Then, since C_0 is well annotated, we have $\Gamma\ \{R'[\mathbf{skip}]\}\ \Gamma'$ for some Γ', and hence we have for computation (iv) that there exists such a β satisfying $\forall c \not\sqsupseteq b$, $\alpha =_c \beta$, as required. □

4 Erasure by Typing

In this section we use the results of the previous section to design a type system for erasure (and noninterference). The idea is that we use Theorem 1 to guide us in the treatment of the input erasure command, standard subject reduction and noninterference properties of the type system to establish a well-annotated version of the program, and Theorem 2 to prove that the type system guarantees input erasure.

Our type system is a simple extension of the flow sensitive system of [HS06] (alternative flow sensitive base systems, such as [AB04], could also be considered). We modify the system of [HS06] to be *termination sensitive*: the rules only allow while loops to be performed over the lowest security level ($\bot$), and these can only occur in the context $\bot$. This is of course a rather restrictive notion. A more liberal system would allow high loops when they can be shown to be terminating.

The type rules are shown in Figure 3. For a command C, judgements have the form $p \vdash \Gamma\ \{C\}\ \Gamma'$ where $p \in \mathcal{L}$, and Γ, Γ' are security type assignments. The idea is that if Γ gives the security levels of variables before execution of C, then Γ' will give their security levels afterwards. The type p represents the usual "program counter" level and serves to eliminate indirect information flows: the rules ensure that only variables with final types (in Γ') greater than or equal to p may be changed by C. Similarly, input and output is only permitted on channels greater than or equal to p.

The purpose of the type system is to guarantee noninterference and input erasure. Here we provide explanation of the rules for input and output, since they are the new ones. The rule for input commands follows Theorem 1 rather directly, making use of a command transformer deleteOutput(C) which simply replaces every output command in its argument with $\mathbf{skip}$. This is the means by which we ignore outputs when checking the local erasure requirement. We cannot however ignore inputs, since we still need to ensure that there are no covert channels via the input pointers. Output is simply treated like an assignment to a variable of a fixed security type. One can note that if we specialise the typing rules to "vanilla" inputs, as represented by commands of the form

$\mathbf{input}\ x : a \nearrow a\ \mathbf{in\ skip}$, then we get what appears to be a flow sensitive version of the deterministic part of the type system from [OCC06].

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

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

$$\text{Output}\,\frac{\Gamma \vdash E : b \qquad p \sqcup b \sqsubseteq a}{p \vdash \Gamma\ \{\mathbf{output}\ E\ \mathbf{on}\ a\}\ \Gamma} \qquad \text{Annotate}\,\frac{p \vdash \Gamma\ \{C\}\ \Gamma'}{p \vdash \Gamma\ \{C^{\Gamma'}\}\ \Gamma'}$$

$$\text{Seq}\,\frac{p \vdash \Gamma\ \{C_1\}\ \Gamma' \quad p \vdash \Gamma'\ \{C_2\}\ \Gamma''}{p \vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma''} \qquad \text{If}\,\frac{\Gamma \vdash E : t \quad p \sqcup t \vdash \Gamma\ \{C_i\}\ \Gamma' \quad i = 1, 2}{p \vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ \Gamma'}$$

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

Fig. 3. Type System

Example Let us reconsider the credit-card transaction server loop from the introduction. Let us suppose that $\bot \sqsubseteq \mathit{user} \sqsubseteq \mathit{bank} \sqsubseteq \top$. To represent the intention that the credit card data is erased by the end of each loop iteration, the code can be rewritten as shown to the right. For the purpose of typing we assume that $\mathit{process}(\mathit{cc})$ is just some expression involving cc. Since $\top$ is used to model the level of data that is no longer physically present, no variables should be given a final type of $\top$. With this restriction there is (thankfully) no typing for this program. The body of the erasure statement C is, in fact, suitably noninterfering, as shown by the typing $\bot \vdash \Gamma\ \{C\}\ \Gamma$ where $\Gamma(\mathit{serverUp}) = \bot$ and $\Gamma(x) = \mathit{user}$ for all other variables x. But to type the enclosing erasure input we also need the typing $\bot \vdash \Gamma[\mathit{cc} \mapsto \top]\ \{\text{deleteOutput}(C)\}\ \Gamma$. This is not possible because $\mathit{payment} := \mathit{process}(\mathit{cc})$ forces $\mathit{payment}$ to type $\top$ instead of user. By appending $\mathit{payment} := 0$ to the end of C the program becomes typeable.

$$\begin{array}{l}
\mathbf{while}\ \mathit{serverUp}\ \{ \\
\quad \mathbf{input}\ \mathit{cc} : \mathit{user} \nearrow \top\ \mathbf{in}\ \{ \\
\quad\quad \left.\begin{array}{l}
\mathbf{input}\ \mathit{details}\ \mathbf{from}\ \mathit{user} \\
\mathit{payment}\ := \ \mathit{process}(\mathit{cc}) \\
\mathbf{output}\ \mathit{payment}\ \mathbf{to}\ \mathit{bank} \\
\mathit{custInfo}\ := \ \mathit{custInfo} \oplus \mathit{details} \\
\mathit{cc}\ := 0
\end{array}\right\} C \\
\quad \} \\
\} \ldots
\end{array}$$

4.1 Type Correctness

In this section we prove correctness. In what follows, we say that C is *well-typed* if, for some p, Γ, Γ', there exists a derivation of $p \vdash \Gamma\ \{C\}\ \Gamma'$.

Before verifying the motivating semantic properties of the type system, we show that it is well behaved with respect to reduction by establishing the obvious subject reduction property.

Theorem 3 (Subject Reduction). *If* $p \vdash \Gamma\ \{C\}\ \Gamma'$ *and* $I \vdash \langle C, s, \mathbf{i}\rangle \twoheadrightarrow \langle C', s', \mathbf{i}'\rangle$*, then* $p \vdash \Gamma''\ \{C'\}\ \Gamma'$*, for some* Γ''*.*

Proof. We prove the property for single-step reductions. This extends immediately to multi-step reductions by induction on the number of steps.

The proof is by induction on the height of the type derivation and by cases on the last rule used.

Case Annotate: We have $C = D^{\Gamma'}$ and the final derivation step is:

$$\dfrac{p \vdash \Gamma\ \{D\}\ \Gamma'}{p \vdash \Gamma\ \{D^{\Gamma'}\}\ \Gamma'}$$

There are two sub-cases to consider:

1. $D = \mathbf{skip}$. In this case $C' = \mathbf{skip}$ and by Skip we have $p \vdash \Gamma'\ \{\mathbf{skip}\}\ \Gamma'$, hence we may take $\Gamma'' = \Gamma'$.
2. $D = R[D_0]$. In this case $C' = R[D_0']^{\Gamma'}$ and the reduction step is justified by:

$$\dfrac{\langle D_0, s, \mathbf{i}\rangle \xrightarrow{\ell} \langle D_0', s', \mathbf{i}'\rangle}{\langle R[D_0]^{\Gamma'}, s, \mathbf{i}\rangle \xrightarrow{\ell} \langle R[D_0']^{\Gamma'}, s', \mathbf{i}'\rangle}$$

Thus:

$$\dfrac{\langle D_0, s, \mathbf{i}\rangle \xrightarrow{\ell} \langle D_0', s', \mathbf{i}'\rangle}{\langle R[D_0], s, \mathbf{i}\rangle \xrightarrow{\ell} \langle R[D_0'], s', \mathbf{i}'\rangle}$$

By IH (applied to the Annotate premise $p \vdash \Gamma\ \{R[D_0]\}\ \Gamma'$), we have $p \vdash \Gamma''\ \{R[D_0']\}\ \Gamma'$ for some Γ''. Hence, by Annotate, $p \vdash \Gamma''\ \{R[D_0']^{\Gamma'}\}\ \Gamma'$, as required.

Case While: We have $C = \mathbf{while}\ E\ D$ and $C' = \mathbf{if}\ E\ (D\ ;\mathbf{while}\ E\ D)$ and the final derivation step is:

$$\text{While}\;\dfrac{\Gamma \vdash E : \bot \qquad \bot \vdash \Gamma\ \{D\}\ \Gamma}{\bot \vdash \Gamma\ \{\mathbf{while}\ E\ D\}\ \Gamma}$$

The required type derivation for C' is constructed as follows:

$$\text{If}\;\dfrac{\Gamma \vdash E : \bot \qquad \text{Seq}\;\dfrac{\bot \vdash \Gamma\ \{D\}\ \Gamma \qquad \bot \vdash \Gamma\ \{C\}\ \Gamma}{\bot \sqcup \bot = \bot \vdash \Gamma\ \{D\ ;C\}\ \Gamma} \qquad \text{Skip}\;\dfrac{}{\bot \vdash \Gamma\ \{\mathbf{skip}\}\ \Gamma}}{\bot \vdash \Gamma\ \{\mathbf{if}\ E\ (D\ ;\mathbf{while}\ E\ D)\}\ \Gamma}$$

Case Erase: We have $C = \mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ D$ and the final derivation step is:

$$\dfrac{p \vdash \Gamma[x \mapsto a]\ \{D\}\ \Gamma' \qquad p \vdash \Gamma[x \mapsto b]\ \{\text{deleteOutput}(C)\}\ \Gamma' \qquad p \sqsubseteq a}{p \vdash \Gamma\ \{\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ D\}\ \Gamma'}$$

In this case $C' = D$ and the result is immediate by the first premise to the final derivation step.

Case Assign: We have $C = x := E$, $\Gamma' = \Gamma[x \mapsto p \sqcup t]$ and the final derivation step is:

$$\frac{\Gamma \vdash E : t}{p \vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup t]}$$

In this case $C' = \mathbf{skip}$ and by Skip we have $p \vdash \Gamma'\ \{\mathbf{skip}\}\ \Gamma'$, hence we may take $\Gamma'' = \Gamma'$.

The remaining cases are similar. □

The two fundamental semantic properties we require of the type system are:

NI Type Correctness: that it guarantees the stateful input-output NI property, Definition 5 (and thus the top level input-output NI property, Definition 3).

Erasure Type Correctness: that it can be used to establish the premises of Theorem 2 (and thus to guarantee input erasure).

Theorem 4 (NI Type Correctness). *If* $p \vdash \Gamma\ \{C\}\ \Gamma'$ *then* $\Gamma\ \{C\}\ \Gamma'$.

Corollary 1. *Well-typed programs are input-output noninterfering.*

The proof of the theorem relies on the following three lemmas.

Lemma 1. *If* $p \neq \bot$ *and* $p \vdash \Gamma\ \{C\}\ \Gamma'$, *then* $I \vdash \langle C, s, \mathbf{i}\rangle \overset{\alpha}{\twoheadrightarrow} \langle \mathbf{skip}, s', \mathbf{i}'\rangle$.

Proof. By inspection of the type system, if $p \vdash \Gamma\ \{C\}\ \Gamma'$ then all sub-commands of C are typeable, each for some $p' \geq p$. Since $\mathbf{while}\ E\ C$ is only typeable for $p' = \bot$ it follows that C contains no loops. By inspection of the transition rules, it is clear that evaluation of C must terminate in $\mathbf{skip}$. (We note that this relies on the fact that expression evaluation is assumed total and that no "stuck" configurations exist for the given semantics.)

Lemma 2. *If* $p \vdash \Gamma\ \{C\}\ \Gamma'$ *and* $p \not\sqsubseteq c$ *then* $\Gamma \vdash s =_c t \Rightarrow \Gamma' \vdash s =_c t$.

Proof. By induction on the height of the type derivation and by cases on the last rule used, it is readily proved that $p \not\sqsubseteq \Gamma'(x) \Rightarrow \Gamma(x) \sqsubseteq \Gamma'(x)$. The lemma follows since, under assumption $p \not\sqsubseteq c$, we then have $\Gamma'(x) \sqsubseteq c \Rightarrow p \not\sqsubseteq \Gamma'(x) \Rightarrow \Gamma(x) \sqsubseteq \Gamma'(x) \Rightarrow \Gamma(x) \sqsubseteq c$; thus $\Gamma \vdash s =_c t \Rightarrow \Gamma' \vdash s =_c t$.

Lemma 3. *If* $p \vdash \Gamma\ \{C\}\ \Gamma'$ *and* $I \vdash \langle C, s, \mathbf{i}\rangle \overset{\alpha}{\twoheadrightarrow} \langle C', s', \mathbf{i}'\rangle$ *then* $c \not\sqsupseteq p \Rightarrow \mathbf{i}' =_c \mathbf{i} \wedge \alpha =_c \epsilon \wedge \Gamma' \vdash s' =_c s$.

Proof. We prove the property for single-step reductions $I \vdash \langle C, s, \mathbf{i}\rangle \xrightarrow{\ell} \langle C', s', \mathbf{i}'\rangle$. This extends to multi-step reductions by induction on the number of steps, using Subject Reduction.

The proof is by induction on the height of the type derivation and by cases on the last rule used. We show the case for Erase by way of illustration.

Case Erase: We have $C = \mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ D$ and the final derivation step is:

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

In this case: $C' = D$, $\ell = a?v$, $s' = s[x \mapsto v]$, $\mathbf{i}' = \mathbf{i}[a \mapsto \mathbf{i}(a) + 1]$.
Since $p \sqsubseteq a$ and $p \not\sqsubseteq c$ it follows that $a \not\sqsubseteq c$. From $a \not\sqsubseteq c$ we have $a?v =_c \epsilon$ and $\mathbf{i}' =_c \mathbf{i}$. From $a \not\sqsubseteq c$ we also have $\mathit{\Gamma}[x \mapsto a] \vdash s =_c s'$ and thus, applying Lemma 2 to the first premise of Erase, $\mathit{\Gamma}' \vdash s =_c s'$, as required. □

Proof (of Theorem 4). The proof is by induction on the height of the type derivation and by cases on the last rule used.

Case If: We have $C = \mathbf{if}\ E\ C_{\mathbf{true}}\ C_{\mathbf{false}}$ and the final derivation step is:

$$\frac{\mathit{\Gamma} \vdash E : a \qquad p \sqcup a \vdash \mathit{\Gamma}\ \{C_v\}\ \mathit{\Gamma}' \quad v \in \{\mathbf{true}, \mathbf{false}\}}{p \vdash \mathit{\Gamma}\ \{\mathbf{if}\ E\ C_{\mathbf{true}}\ C_{\mathbf{false}}\}\ \mathit{\Gamma}'}$$

Assume $I =_c I'$, $\mathit{\Gamma} \vdash s =_c t$, $\mathbf{i} =_c \mathbf{j}$. Now, assume

$$I \vdash \langle C, s, \mathbf{i} \rangle \overset{\alpha}{\twoheadrightarrow} \langle C', s', \mathbf{i}' \rangle$$

In this case $\alpha = \tau\alpha'$ with

$$I \vdash \langle C, s, \mathbf{i} \rangle \xrightarrow{\tau} \langle C_v, s, \mathbf{i} \rangle \overset{\alpha'}{\twoheadrightarrow} \langle C', s', \mathbf{i}' \rangle$$

where $v = [\![E]\!]s$, and

$$I' \vdash \langle C, t, \mathbf{j} \rangle \xrightarrow{\tau} \langle C_w, t, \mathbf{j} \rangle$$

where $w = [\![E]\!]t$. We must show:

1. $\langle C_w, t, \mathbf{j} \rangle \overset{\beta}{\twoheadrightarrow} \langle C'', t', \mathbf{j}' \rangle$ with $\beta =_c \alpha'$.
2. If $C' = \mathbf{skip}$ then $\langle C_w, t, \mathbf{j} \rangle \twoheadrightarrow \langle \mathbf{skip}, t', \mathbf{j}' \rangle$ with $t' =_c s'$ and $\mathbf{j}' =_c \mathbf{j}$.

There are two sub-cases to consider:
Case $v = w$: Both parts follow by IH applied to $p \sqcup a \vdash \mathit{\Gamma}\ \{C_v\}\ \mathit{\Gamma}'$.
Case $v \neq w$: In this case, $a \not\sqsubseteq c$ (since, otherwise, $\mathit{\Gamma} \vdash s =_a t$ hence $[\![E]\!]s = [\![E]\!]t$), thus $p \sqcup a \not\sqsubseteq c$. Applying Lemma 1 to the typing premise for C_w gives $\langle C_w, t, \mathbf{j} \rangle \overset{\beta'}{\twoheadrightarrow} \langle \mathbf{skip}, t', \mathbf{j}' \rangle$ and applying Lemma 3 to both premises gives $\alpha' =_c \epsilon =_c \beta$. Applying Lemma 2 to either premise (plus assumption $\mathit{\Gamma} \vdash s =_c t$) gives $\mathit{\Gamma}' \vdash s =_c t$. By assumption, $\mathbf{i} =_c \mathbf{j}$. Applying Lemma 3 to both premises gives $\mathbf{i}' = \mathbf{i}$, $\mathbf{j} = \mathbf{j}'$, $\mathit{\Gamma}' \vdash s' =_c s$ and $\mathit{\Gamma}' \vdash t =_c t'$. Hence $\mathbf{i}' =_c \mathbf{j}'$ and $\mathit{\Gamma}' \vdash s' =_c t'$ follow by transitivity of $=_c$.

Case Erase: We have $C = \mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ D$ and the final derivation step is:

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

Assume $I =_c I'$, $\mathit{\Gamma} \vdash s =_c t$, $\mathbf{i} =_c \mathbf{j}$. Now, assume

$$I \vdash \langle C, s, \mathbf{i} \rangle \overset{\alpha}{\twoheadrightarrow} \langle C', s', \mathbf{i}' \rangle$$

In this case $\alpha = a?v\alpha'$ with

$$I \vdash \langle C, s, \mathbf{i}\rangle \xrightarrow{a?v} \langle D, s[x \mapsto v], \mathbf{i}[a \mapsto n]\rangle \overset{\alpha'}{\twoheadrightarrow} \langle C', s', \mathbf{i}'\rangle$$

(where $n = \mathbf{i}(a) + 1$ and $v = I_a(n)$) and

$$I' \vdash \langle C, t, \mathbf{j}\rangle \xrightarrow{a?v'} \langle D, t[x \mapsto v'], \mathbf{j}[a \mapsto n']\rangle$$

(where $n' = \mathbf{j}(a) + 1$ and $v' = I'_a(n')$).
Now if $a \sqsubseteq c$, then by the assumptions $n = n'$ and $v' = v$, hence $a?v' =_c a?v$ and $\Gamma[x \mapsto a] \vdash s[x \mapsto v] =_c t[x \mapsto v']$ and $\mathbf{i}[a \mapsto n] =_c \mathbf{j}[a \mapsto n']$. On the other hand, if $a \not\sqsubseteq c$ then $a?v' =_c \epsilon =_c a?v$ and $\Gamma[x \mapsto a] \vdash s[x \mapsto v] =_c t[x \mapsto v']$ and $\mathbf{i}[a \mapsto n] =_c \mathbf{j}[a \mapsto n']$, regardless of the values v, v'. Stateful NI then follows by IH applied to the typing premise $p \vdash \Gamma[x \mapsto a]\ \{D\}\ \Gamma'$.

Case Assign: In this case C is $x := E$, $\Gamma' = \Gamma[x \mapsto p \sqcup a]$ and the final derivation step is:

$$\frac{\Gamma \vdash E : a}{p \vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup a]}$$

Assume $I =_c I'$, $\Gamma \vdash s =_c t$, $\mathbf{i} =_c \mathbf{j}$. We have

$$I \vdash \langle C, s, \mathbf{i}\rangle \xrightarrow{\tau} \langle \mathbf{skip}, s[x \mapsto v], \mathbf{i}\rangle$$

where $v = [\![E]\!]s$, and

$$I \vdash \langle C, t, \mathbf{j}\rangle \xrightarrow{\tau} \langle \mathbf{skip}, t[x \mapsto w], \mathbf{j}\rangle$$

where $w = [\![E]\!]t$. It suffices then to show

$$\Gamma[x \mapsto p \sqcup a] \vdash s[x \mapsto v] =_c t[x \mapsto w] \quad (*)$$

Now if $a \sqsubseteq c$, then $\Gamma \vdash s =_c t$ implies $\Gamma \vdash s =_a t$, hence $v = w$, hence $(*)$ holds. On the other hand, if $a \not\sqsubseteq c$ then $(*)$ follows from $\Gamma \vdash s =_a t$, regardless of the values v, w.

Case Seq: In this case $C = C_1\ ;\ C_2$ and the final derivation step is:

$$\frac{p \vdash \Gamma\ \{C_1\}\ \Gamma' \qquad p \vdash \Gamma'\ \{C_2\}\ \Gamma''}{p \vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma''}$$

It is easily verified that any derivation $I \vdash \langle C, s, \mathbf{i}\rangle \overset{\alpha}{\twoheadrightarrow} \langle C', s', \mathbf{i}'\rangle$ has one of the two following forms:

1. $I \vdash \langle C_1\ ;\ C_2, s, \mathbf{i}\rangle \overset{\alpha}{\twoheadrightarrow} \langle C'_1\ ;\ C_2, s', \mathbf{i}'\rangle$ where $I \vdash \langle C_1, s, \mathbf{i}\rangle \overset{\alpha}{\twoheadrightarrow} \langle C'_1, s', \mathbf{i}'\rangle$
2. $I \vdash \langle C_1\ ;\ C_2, s, \mathbf{i}\rangle \overset{\alpha}{\twoheadrightarrow} \langle C'_2, s', \mathbf{i}'\rangle$ where $I \vdash \langle C_1, s, \mathbf{i}\rangle \overset{\alpha'}{\twoheadrightarrow} \langle \mathbf{skip}, s'', \mathbf{i}''\rangle$ and $I \vdash \langle C_2, s'', \mathbf{i}''\rangle \overset{\alpha''}{\twoheadrightarrow} \langle C'_2, s', \mathbf{i}'\rangle$, with $\alpha = \alpha'\tau\alpha''$.

In either case the result follows straightforwardly by application of IH to the premises of the type derivation.

The remaining cases are similar. □

Theorem 5 (Erasure Type Correctness). *If C is well-typed then C is well-annotated.*

Corollary 2. *Well-typed programs are input erasing.*

Proof. By inspection of the type system, any derivation of a typing for a program must include a sub-derivation $p \vdash \Gamma\ \{\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C\}\ \Gamma'$ for every input command, and we can use each such Γ' to annotate the corresponding input command. By inserting uses of Annotate into the original type derivation we can clearly recover a derivation for the annotated program. By Theorem 5 the annotated program is well-annotated and hence, by Theorem 2, is input erasing. Since the annotated program is semantically equivalent to the original, it follows that the original is input erasing. □

Proof (of Theorem 5). The proof of the theorem is in two parts, corresponding to the two parts of the definition of well-annotation. For the first part we rely on Theorem 1, which shows that well-annotation of input commands is a corollary of the following lemma:

Lemma 4. *If $p \vdash \Gamma\ \{(\mathbf{input}\ x : a \nearrow b\ \mathbf{in}\ C)^{\Gamma'}\}\ \Gamma''$ then $\Gamma[x \mapsto b]\ \{\widehat{C}\}\ \Gamma'$.*

For the second part, we rely on the following lemma:

Lemma 5. *If $p \vdash \Gamma_0\ \{R[\mathbf{skip}^{\Gamma}]\}\ \Gamma'$ then $\Gamma\ \{R[\mathbf{skip}]\}\ \Gamma'$.*

The second part of well-annotation then follows by subject reduction. □

The proofs of the lemmas follow.

Proof (of Lemma 4). Assume lhs. By inspection of the type system, we have a sub-derivation $p_1 \vdash \Gamma_1[x \mapsto b]\ \{\text{deleteOutput}(C)\}\ \Gamma_1'$, with $p \sqsubseteq p_1$, $\Gamma \sqsubseteq \Gamma_1$, $\Gamma_1' \sqsubseteq \Gamma'$. By Theorem 4, $\Gamma_1[x \mapsto b]\ \{\text{deleteOutput}(C)\}\ \Gamma_1'$. Since $\Gamma \sqsubseteq \Gamma_1$, it follows that $\Gamma[x \mapsto b] \sqsubseteq \Gamma_1[x \mapsto b]$, hence $\Gamma[x \mapsto b]\ \{\text{deleteOutput}(C)\}\ \Gamma'$, by montonicity. It is clear that the behaviours of deleteOutput(C) and $\widehat{C}$ are identical except that any non-τ event labels on the transitions of deleteOutput(C) are replaced by τ on the transitions of $\widehat{C}$. It follows that $\Gamma[x \mapsto b]\ \{\widehat{C}\}\ \Gamma'$, as required. □

Proof (of Lemma 5). By induction on the structure of $R[\cdot]$.

Case $R[\cdot] = [\,\cdot\,]$**:** We have a derivation $p \vdash \Gamma_0\ \{\mathbf{skip}^{\Gamma}\}\ \Gamma'$ and we must show $\Gamma\ \{\mathbf{skip}\}\ \Gamma'$, which will follow if $\Gamma \sqsubseteq \Gamma'$. This latter is easily seen to hold because the given derivation must end with a single use of Annotate followed by zero or more uses of Sub.

Case $R[\cdot] = R'[\cdot]\ ; C'$**:** We have a derivation $p \vdash \Gamma_0\ \{R'[\mathbf{skip}^{\Gamma}]; C'\}\ \Gamma'$. This derivation must end with Seq followed by zero or more uses of Sub, hence we have derivations $p' \vdash \Gamma_0'\ \{R'[\mathbf{skip}^{\Gamma}]\}\ \Gamma''$ and $p' \vdash \Gamma''\ \{C'\}\ \Gamma'''$ with $\Gamma''' \sqsubseteq \Gamma'$. By IH $\Gamma\ \{R'[\mathbf{skip}]\}\ \Gamma''$ and by Theorem 4 $\Gamma''\ \{C'\}\ \Gamma'''$. Hence, by compositionality and monotonicity, $\Gamma\ \{R'[\mathbf{skip}]; C'\}\ \Gamma'$, as required.

Case $R[\cdot] = R'[\cdot]^{\Gamma''}$**:** We have a derivation $p \vdash \Gamma_0\ \{R'[\mathbf{skip}^{\Gamma}]^{\Gamma''}\}\ \Gamma'$. This derivation must end with Annotate followed by zero or more uses of Sub, hence we have derivation $p \vdash \Gamma_1\ \{R'[\mathbf{skip}^{\Gamma}]\}\ \Gamma''$ with $\Gamma_0 \sqsubseteq \Gamma_1$, $\Gamma'' \sqsubseteq \Gamma'$. By IH $\Gamma\ \{R'[\mathbf{skip}]\}\ \Gamma''$ hence, by monotonicity, $\Gamma\ \{R'[\mathbf{skip}]\}\ \Gamma'$. Since the operational semantics of annotations are transparent, it follows that $\Gamma\ \{R'[\mathbf{skip}]^{\Gamma''}\}\ \Gamma'$, as required. □

5 On the Adequacy of the Input-Output Model

We have adopted a simple stream-based model of input-output. In a general nondeterministic setting, such a model does not adequately model a “high” attacker who is trying to pass information to “low” through the program, and it becomes necessary to quantify over all possible *strategies* adopted by the principals. This is a well known problem in the noninterference literature [WJ90]. See [OCC06] for a recent language-based take on the issue. Fortunately, since we deal with deterministic programs, it turns out that simple stream models *are* nevertheless adequate, as shown recently by Clark and Hunt [CH07].

What about erasure? Are there potential problems that arise from not modelling an active attacker’s strategy? In fact the problem here is that we *cannot* reasonably model inputs as coming from an attacker with an arbitrary strategy, because it only makes sense to promise to erase data if the supplier is *not* an adversary. A payment system typically promises, on completion of a transaction, to erase the credit card data but to retain the shipping address. The system will not succeed in erasing the credit card data if the user’s strategy is to re-input the credit card data as a response to a subsequent request for the shipping address, but clearly we do not want to admit such strategies.

There are more subtle cases which show that we must assume even more about the data supplier’s behaviour. Suppose that, before the credit card is erased, the program sends back to the user a special offer code *“zahojasf23”* with the promise “present this code when you next shop with us for a 10% discount”. What if this code is simply an encryption of the credit card number? The program in this case may well have erased the credit card number by the end of the transaction, but if the user re-inputs this code then the program will have reconstructed the credit card number.

What assumptions are reasonable for the data supplier? We assume, from a noninterference perspective, that attackers can make arbitrarily accurate semantic deductions based on their observations and complete knowledge of the program. For a non attacker it seems reasonable to assume the opposite – the honest user sees the program as a black box. How then can we solve the problem from the example above if the user cannot be relied upon to *know* whether *“zahojasf23”* contains their credit card information? Our proposed solution is to:

- assume that the user is aware of the erasure “contract”; they know that they are providing an input which is scheduled for erasure, and they are notified when the erasure is complete, and
- assume that the user treats any outputs from the program (at their level) as potentially tainted with data currently scheduled for erasure.

We believe that the stream model that we have used here correctly captures these assumptions, but it is beyond the scope of this paper to explicitly model such user strategies in order to *prove* that the stream model is indeed correct in this sense.

6 Conclusions and Further Work

We have studied the semantics of erasure and shown its connection to noninterference. We have introduced a particular idiom for expressing erasure policies in code, and shown that a natural global erasure property can be enforced by a combination of noninterference and a local erasure property, which in turn can also be established by a noninterference property. This leads to a fairly direct definition of a type system for which well typed programs correctly erase their data. We conclude here by returning to the related work, before finishing with some remarks about further work.

Related work In addition to Chong and Myers work [CM05], Hansen and Probst [HP06] describe what they call *simple erasure policies* which correspond to a specific instance of our end-to-end erasure policies, but stated in terms of the erasure of a whole level rather than a single variable. Neither of these works describe an implementation of erasure, either by encoding into standard noninterference or developing a specific program analysis.

There are several fundamental differences between the *definition* of erasure developed here and that of Chong and Myers. Ignoring the fact that [CM05] also deals with declassification policies, we note the following differences. Firstly, [CM05] does not consider a system with interaction, something that we feel is central to making notions of erasure meaningful. Secondly, in the abstract system model in [CM05] the state of the system is *just* a store. The obvious way to encode an imperative program as such a system would be to use a *program counter* variable, but there is no suitable *policy* in their language which one could attach to such a program counter. Thus their model might not be suitable for modelling imperative programs – at least not with a straightforward encoding. Thirdly, they require a "physical erasure" condition which says that at the point where a variable is erased it should contain a predefined constant. This is stronger than necessary. Although we can *satisfy* erasure properties in that way, there is nothing to stop us from erasing data to level b by e.g. overwriting it with something else from a lower level. Lastly, since erasure can be thought of as a dual to declassification (since it is used to strengthen as opposed to weaken NI) we can see that their erasure condition and ours tackle different *dimensions* of erasure: using the terminology of [SS05], their erasure properties deal with *when* erasure takes place, whereas our input-centric erasure determines *where* (in the code) erasure takes place.

Finally, we note that our use of a block structured erasure command is similar in spirit to Almeida Matos and Boudol's [AB05] block structured declassification construct, $\mathbf{flow}\ F\ \mathbf{in}\ C$, which locally extends the global information flow policy with flows F for the duration of C.

Further Work There are several obvious avenues for further work.

We can follow the "dimensions" and consider, for example, refinement of *what* is erased. For example, erasure of all except the first four digits of a credit card number. Work on corresponding "what" declassification policies [SS05] can be applied directly.

The input erasure construct used here can be generalised in a number of potentially useful ways. One possibility is to introduce an erasure region – a code block in which all subsequent inputs are erased.

A naive implementation of the type system as presented is potentially exponential in the depth of nesting of erasure statements, because the body of the erasure statement appears twice in the premise of the Erase rule. By building on results from [HS06], we are hopeful that this behaviour can be avoided by obtaining the two typings for the body of an erasure input as specialisations of a single principal type.

On the theoretical side we noted at the end of the previous section the need for further work on modelling attacker strategies and "honest" participants. A process calculus setting may prove more suitable to conduct such an investigation.

Acknowledgements Thanks to various members of the ProSec group at Chalmers for helpful comments, to Steve Chong and to the anonymous referees for numerous helpful comments and suggestions. This work was partly supported by EPSRC research grant EP/C009746/1 Quantitative Information Flow, the Swedish research agencies Vinnova, SSF, VR and by the Information Society Technologies programme of the European Commission under the IST-2005-015905 MOBIUS project.

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.

AB05. A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

CH07. D. Clark and S. Hunt. Observation, nondeterminism and nondeducability on strategies. Workshop presentation at PLID'07, 3rd International Workshop on Programming Language Dependence and Independence, August 2007.

CM05. S. Chong and A. C. Myers. Language-based information erasure. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

FG95. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. Computer Security*, 3(1):5–33, 1995.

GM04. R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 186–197, 2004.

HP06. R. R. Hansen and C. W. Probst. Non-interference and erasure policies for java card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS '06)*, 2006.

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.

OCC06. Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201. IEEE Computer Society, 2006.

Ros95. A. W. Roscoe. CSP and determinism in security modeling. In *Proc. IEEE Symp. on Security and Privacy*, pages 114–127, May 1995.

SS01. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

SS05. Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society Press, 2005.

WJ90. J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *IEEE Symposium on Security and Privacy*, pages 144–161, 1990.