End-to-End Enforcement of Erasure and Declassification

Stephen Chong   Andrew C. Myers

Department of Computer Science
Cornell University
{schong,andru}@cs.cornell.edu

Computing and Information Science Technical Report
http://hdl.handle.net/1813/10504

April 2007
Revised June 2008

End-to-End Enforcement of Erasure and Declassification

Stephen Chong Andrew C. Myers
Department of Computer Science
Cornell University
{schong,andru}@cs.cornell.edu

Abstract

Declassification occurs when the confidentiality of information is weakened; *erasure* occurs when the confidentiality of information is strengthened, perhaps to the point of completely removing the information from the system.

This paper shows how to enforce erasure and declassification policies. A combination of a type system that controls information flow and a simple runtime mechanism to overwrite data ensures end-to-end enforcement of policies. We prove that well-typed programs satisfy the semantic security condition *noninterference according to policy*.

We extend the Jif programming language with erasure and declassification enforcement mechanisms and use the resulting language in a large case study of a voting system.

1 Introduction

Enforcing information security is an important requirement of many systems. However, often information security changes over time, complicating enforcement. *Declassification* and *erasure* are two common ways in which the security enforced on information changes. Declassification occurs when the confidentiality enforced on information is weakened, for example, by allowing more people to read the information. Erasure [2] is the opposite phenomenon, occurring when the confidentiality enforced on information is strengthened, perhaps to the point of removing the information from the system entirely.

Much work in recent years has considered how to provide end-to-end enforcement of declassification requirements. (See Sabelfeld and Sands [21] for a recent survey.) Comparatively little work [11] has considered end-to-end enforcement of erasure requirements, and none has considered both declassification and erasure together. In this paper, we enforce both erasure and declassification requirements end-to-end in a language-based setting. The erasure policies we enforce are significantly more expressive than any previously enforced.

This work was supported by National Science Foundation grant 0430161. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of these organizations or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Consider, as an example of erasure requirements, a medical information website. The website offers (among other functionality) a diagnostic application, where a user may enter information about symptoms, and the application will present information about possible diseases consistent with the symptoms. The website's privacy policy states that symptoms the user enters are private, and no record of them will be kept after the user has finished using the diagnostic application. The provider of this website needs to enforce an erasure requirement: when the user has finished using the diagnostic application, the symptom data that the user has entered must be erased. Note that the information the user has entered may need to persist over several user requests, but also might need to be erased before the session has finished. Thus, the lifetime of the information does not necessarily match that of any web server resource. Another subtlety is that the diagnoses the system has produced must also be erased, as the diagnoses may reveal information about the symptoms entered.

Information security is an end-to-end requirement: information security policies must be enforced on information no matter how it propagates through the system or where it enters or leaves. These policies should also be enforced on data derived from sensitive information, since derived data may allow deductions about source information. In the diagnostic application described above, learning diagnoses may reveal symptoms, which are sensitive.

Information-flow control is an approach for achieving end-to-end enforcement. Information-flow control techniques enforce security by restricting the flow, or propagation, of information in a system. Conceptually, information-flow control techniques *label* data with security levels; as data are updated and created, the security labels are also updated to reflect data dependencies. The security labels can be used to prevent confidential data from being output on public channels. Static enforcement methods [20] can control information flow without incurring the performance overheads of representing security labels at runtime.

In this paper, we use static information-flow control to enforce erasure and declassification requirements end-to-end. Erasure and declassification requirements are specified in a policy language that can express when information *may* be declassified, and when information *must* be erased. Section 2 reviews this policy language, which was introduced in earlier work [2] but lacked any enforcement mechanism.

a, b	Conditions
$p, q ::=$	Policies
$\quad \ell$	Lattice policy
$\quad p \overset{a}{\searrow} q$	Declassification policy
$\quad p \;{}^{a}\!\!\nearrow q$	Erasure policy

Figure 1: Syntax of policies

Section 3 presents a simple imperative language that has a runtime mechanism for overwriting memory locations. A type system controls information flow, ensuring that information that needs to be erased is placed only in memory locations that will be overwritten at appropriate times.

Noninterference [6] is an end-to-end semantic security condition. It is well-known that noninterference is too strong in the presence of declassification, which intentionally makes sensitive information public. However, noninterference is too weak in the presence of erasure—it cannot express erasure requirements, which restrict observation of public information. Section 4 proves that well-typed programs satisfy *noninterference according to policy* [2], a generalization of noninterference that precisely expresses the information flows permitted by declassification and erasure.

Section 5 describes how we incorporated declassification and erasure policies into the decentralized label model [16] and extended the Jif programming language [17] with the new label model. We have used this extended version of Jif to implement a large, security-intensive system: Civitas [3], a secure voting service. Section 6 describes how the policies are useful in its implementation. Section 7 reviews related work, and Section 8 concludes.

2 Policies

The declassification and erasure policy framework introduced in previous work [2] assumes there is a lattice $(\mathcal{L}, \sqsubseteq)$ of *confidentiality levels*, and a language for specifying *conditions*, which indicate when declassification may occur and when erasure must occur. To instantiate the policy framework, lattice $\mathcal{L}$ and the condition language must be specified. Appropriate security lattices include the two-point lattice $\{L, H\}$ where $L \sqsubseteq H$ and $H \not\sqsubseteq L$, and the lattice of security principals ordered by an *acts-for* relation [16]. (In Section 5 we use the lattice of security principals when extending the decentralized label model [16].) We assume there is a clear notion of enforcement of confidentiality level $\ell \in \mathcal{L}$ on information. Many condition languages are possible; Section 3 uses program expressions as conditions.

2.1 Syntax

Security policies describe what confidentiality level is currently enforced on information, and how this may and must change in the future. Figure 1 shows the syntax of policies. Lattice policy $\ell \in \mathcal{L}$ means that the confidentiality level ℓ (or a more restrictive confidentiality level) must be enforced on information now and at all times in the future. Declassification policy $p \overset{a}{\searrow} q$ means that policy p is currently enforced on information, and when condition a is satisfied, information may be declassified, after which policy q must be enforced (regardless of the subsequent satisfaction or non-satisfaction of a). Erasure policy $p \;{}^{a}\!\!\nearrow q$ means that policy p is currently enforced on information, and when condition a is satisfied, information must be made more restricted, by enforcing both policies p and q on the information (regardless of the subsequent satisfaction or non-satisfaction of a).

The satisfaction of conditions controls when declassification may occur, and when erasure must occur. Condition satisfaction is specific to the condition language used. We assume condition satisfaction depends only on the current system state s (which may include the history of the system), and write $s \vDash a$ if condition a is satisfied in state s, and $s \nvDash a$ if a is not satisfied in state s.

For example, if we are enforcing policy $H \overset{a}{\searrow} L$ on information, then we must enforce the confidentiality level H on the information; however, when condition a is satisfied, we are permitted to change the confidentiality level enforced on the information to L. If we are enforcing erasure policy $L \;{}^{a}\!\!\nearrow H$ on information, then we must enforce the confidentiality level L on the information, and if and when condition a is satisfied, we must change the confidentiality level we are enforcing to be at least as restrictive as both L and H—since $L \sqsubseteq H$, it suffices to enforce the confidentiality level H.

Consider enforcing policy $(H \overset{a}{\searrow} L) \;{}^{b}\!\!\nearrow H$ on information. Initially policy $H \overset{a}{\searrow} L$ is enforced on information, meaning that the confidentiality level H must be enforced, and if condition a is satisfied (before b is satisfied) then confidentiality level L can be enforced on information. However, once condition b is satisfied, we must enforce policy H on information, meaning that confidentiality level H will be enforced then and at all times in the future.

To see how these policies can capture the security requirements of applications, let us revisit the medical information website example from the introduction. A suitable policy for symptoms entered by the user could be $session \;{}^{appEnd}\!\!\nearrow \top$, where $session$ is a confidentiality level allowing only the session client and server to read the information, $\top$ is a confidentiality level so restrictive that it prevents the server from storing the information, and $appEnd$ is a condition that is satisfied when the user has finished using the diagnosis application. Thus, the data entered by the user will initially have the confidentiality level $session$ enforced on it. Once condition $appEnd$ is satisfied, the confidentiality level $\top$ must be enforced, implying that the data will be removed completely from the system. End-to-end enforcement of the policies will ensure that information derived from the user's symptoms will have the same policy, $session \;{}^{appEnd}\!\!\nearrow \top$ enforced on it, or something more restrictive. Thus, any diagnoses derived from the user's symptoms must also have the confidentiality level $\top$ enforced on them once $appEnd$ is satisfied.

Condition satisfaction determines when policies mandate erasure. Since condition satisfaction is determined solely by the system state, we say that policy p *requires information*

$$\frac{\mathsf{reqErase}(p, s)}{\mathsf{reqErase}(p \searrow^{a} p', s)} \qquad \frac{\mathsf{reqErase}(p, s) \text{ or } s \vDash a}{\mathsf{reqErase}(p \overset{a}{\nearrow} p', s)}$$

Figure 2: Definition of $\mathsf{reqErase}(p, s)$

erasure in state s (or simply, *requires erasure* in state s), denoted $\mathsf{reqErase}(p, s)$, if there is a currently enforced erasure policy whose condition is satisfied. Figure 2 gives inference rules defining $\mathsf{reqErase}(p, s)$. Lattice policy ℓ never requires erasure. Declassification policy $p \searrow^{a} q$ requires erasure if subpolicy p (the policy currently enforced) requires erasure. Erasure policy $p \overset{a}{\nearrow} q$ requires erasure if subpolicy p requires erasure, or a is satisfied. If policy p is enforced on information, then we must ensure that in any state s such that p requires erasure in s, the information either is removed from the system, or has a suitably restrictive policy enforced on it.

2.2 Semantics

Intuitively, the policies describe how the confidentiality of information may and must change as the system executes. We formalize this intuition by providing a semantics for policies.

The semantics of policy p in state s, denoted $[\![p]\!]_s$, is a set of pairs of system states and confidentiality levels that describes what confidentiality levels may be enforced on information labeled p in state s as the system evolves from state s. If policy p is enforced on information in state s, and $(s', \ell') \in [\![p]\!]_s$, then by the time the system reaches state s' (in zero or more steps), confidentiality level ℓ' may be enforced on the information.

$$\begin{aligned}
[\![\ell]\!]_s &= \{(s', \ell') \mid s \rightarrow^* s' \text{ and } \ell \sqsubseteq \ell'\} \\
[\![p \searrow^{a} q]\!]_s &= [\![p]\!]_s \cup \bigcup\{[\![q]\!]_{s'} \mid s \rightarrow^* s' \text{ and } s' \vDash a\} \\
[\![p \overset{a}{\nearrow} q]\!]_s &= [\![p]\!]_s \cap \Big(\{(s', \ell) \in [\![p]\!]_s \mid [s, s'] \nvDash a\} \;\cup \\
&\qquad \bigcup\{[\![q]\!]_{s''} \mid s \rightarrow^* s'' \text{ and } [s, s'') \nvDash a\}\Big)
\end{aligned}$$

Figure 3: Semantics for policies $[\![p]\!]_s$

Figure 3 defines the semantics $[\![p]\!]_s$. We assume that the relation $\rightarrow$ over system states describes atomic transitions of the system, and denote the reflexive transitive closure of this relation as $\rightarrow^*$.

The semantics for confidentiality level ℓ allow any confidentiality level at least as restrictive as ℓ to be enforced at all times in the future.

The semantics of declassification policy $p \searrow^{a} q$ is a superset of the semantics of policy p. The semantics capture the intuition that when the condition is satisfied, the information may be declassified, and after declassification, policy q is enforced on the declassified information. If p permits enforcing confidentiality ℓ in state s', then $p \searrow^{a} q$ also permits it, and in addition, permits policy q to be enforced on information, starting in any state s' such that $s' \vDash a$.

By contrast, the semantics of erasure policy $p \overset{a}{\nearrow} q$ in state s is a subset of the semantics of p in s. The intuition is that policy p is enforced while condition a is not satisfied, and once condition a is satisfied, the information is made more restricted by enforcing both policies p and q. We write $[s, s'] \nvDash a$, where $s \rightarrow^* s'$, to mean that condition a is not satisfied in any state from s to s' inclusive:

$$[s, s'] \nvDash a \triangleq \forall s''.\ (s \rightarrow^* s'' \wedge s'' \rightarrow^* s') \Rightarrow s'' \nvDash a.$$

Similarly, we use $[s, s') \nvDash a$, to mean that condition a is not satisfied in any state from s up to but not including, state s':

$$\begin{aligned}
&[s, s') \nvDash a \triangleq \\
&\qquad \forall s''.\ (s \rightarrow^* s'' \wedge s'' \rightarrow^* s' \wedge s'' \neq s') \Rightarrow s'' \nvDash a.
\end{aligned}$$

2.3 Relabeling judgment

We can define a *relabeling judgment* $a_0, \ldots, a_k \vdash p \leq q$ such that if $a_0, \ldots, a_k \vdash p \leq q$ then, assuming conditions $a_0, \ldots, a_k$ are all satisfied, information labeled with policy p can safely be relabeled with policy q. That is, enforcing q on the information is consistent with policy p. Any such relabeling judgment should be sound with respect to the semantics, and we require the following property to hold.

Property 1 (Soundness) *If $a_0, \ldots, a_k \vdash p \leq q$ then for all states s, such that $\forall i \in 0..k.\ s \vDash a_i$, we have $[\![q]\!]_s \subseteq [\![p]\!]_s$.*

A sound relabeling judgment serves as a syntactic approximation of the policy semantics. Inference rules for a relabeling judgment, and a proof of soundness, are given in Appendix A. In the following section, we use the relabeling judgment in the type system to enforce policies syntactically, without reference to policy semantics.

Other sound syntactic approximations of the policy semantics are possible. In earlier work [2] we introduced a sound relabeling relation parameterized on the current state of the system. This permits reasoning about the subsequent execution of the system, in addition to the conditions satisfied in the current system state.

3 Language

In this section we present a simple imperative language, IMP_E, that incorporates declassification and erasure policies. The language has runtime mechanisms for erasure and declassification, and a type system to control the flow of information. In Section 4, we show that these together suffice to enforce declassification and erasure policies.

3.1 Syntax

Figure 4 presents the syntax of IMP_E. We assume there is a countable set of variables Vars. Language expressions include integer literals $n \in \mathbb{Z}$, and variables $x \in \mathit{Vars}$. The metavariable $\oplus$ ranges over total binary operations on integers.

$e ::=$	Expressions
$\quad n$	Integer literal
$\quad x$	Variable
$\quad e_0 \oplus e_1$	Binary operation
$c ::=$	Commands
$\quad$ **skip**	No-op
$\quad x := e$	Assignment
$\quad c_0; c_1$	Sequence
$\quad$ **if** e **then** c_0 **else** c_1	Selection
$\quad$ **while** e **do** c	Iteration
$\quad x :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$	Guarded declassification

Figure 4: Syntax of IMP_E

Conditions of policies in IMP_E are simply expressions. A condition is satisfied when it evaluates to a non-zero value. For example, if policy $H \searrow^{x+3} L$ is enforced on information, that information may be declassified when expression $x + 3$ is non-zero.

The commands are standard, with the exception of declassification. The *guarded declassification* command $x :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$ evaluates expression e, and assigns the result to variable x, provided that expression e_i evaluates to a non-zero value, for all $0 \leq i \leq k$. If there is some e_i that evaluates to zero, declassification fails. The expressions e_i are conditions that must be satisfied for the declassification to occur. The guarded declassification command allows the type system to check that, assuming all conditions e_i are satisfied, information labeled p_f can safely be relabeled p_t, and allows the operational semantics to ensure that conditions e_i are indeed satisfied when declassification occurs. The type system and runtime mechanisms for enforcing declassification are discussed further in Sections 3.2 and 3.3.

3.2 Operational semantics

A *memory* σ is a map from variables to integers, and is thus a function from Vars to $\mathbb{Z}$. We write $\sigma(e)$ for the result of evaluating expression e using memory σ, that is, using $\sigma(x)$ as the value of each variable x that occurs in e. We write $\sigma[x \mapsto v]$ for the memory that maps variable x to integer v, and otherwise behaves exactly as σ does.

A *configuration* is a pair of a command c and memory σ, written $\langle c, \sigma\rangle$. A configuration fully describes the system state. Since policy conditions are expressions, the satisfaction of a condition depends only on the memory of the current configuration. For brevity, we thus write $\mathsf{reqErase}(p, \sigma)$ instead of $\mathsf{reqErase}(p, \langle c, \sigma\rangle)$.

We assume there is a *typing context* that indicates what policy should be enforced on information stored in each variable. A typing context Γ is a function from Vars to policies, and $\Gamma(x)$ is the policy that must be enforced on information stored in variable x. The typing context does not change during execution: a variable x always has the same policy $\Gamma(x)$

$$\mathit{update}(\sigma, x, v) = \begin{cases} \mathit{erasure}(\sigma) & \text{if } \mathsf{reqErase}(\Gamma(x), \sigma) \\ \mathit{erasure}(\sigma[x \mapsto v]) & \text{otherwise} \end{cases}$$

and

$$\mathit{erasure}(\sigma) = \bigsqcup_{i \in \omega} \sigma_i$$

where $\sigma_0 = \sigma$, and

$$\sigma_{i+1} = \lambda x \in \mathit{Vars}.\ \begin{cases} 0 & \text{if } \mathsf{reqErase}(\Gamma(x), \sigma_i) \\ \sigma_i(x) & \text{otherwise} \end{cases}$$

and $\bigsqcup_{i \in \omega} \sigma_i$ denotes the least upper bound of the chain $\sigma_0\sigma_1\sigma_2 \ldots$ under the ordering $\sqsubseteq$, where

$\sigma' \sqsubseteq \sigma'' \triangleq \forall x \in \mathit{Vars}.\ \sigma'(x) = \sigma''(x) \vee \sigma''(x) = 0$

Figure 6: $\mathit{update}(\sigma, x, v)$ and $\mathit{erasure}(\sigma)$

enforced on it.

Figure 5 presents the operational semantics for IMP_E, showing how configurations are updated as commands execute. The enforcement of policies relies on two runtime mechanisms, embodied in the operational semantics. The first is runtime overwriting of variables to enforce erasure; the second is runtime checking of conditions for declassification. Except for these two mechanisms, the operational semantics of the language are standard.

Overwriting variables. IMP_E enforces erasure by setting the contents of a variable to zero whenever the policy for the variable requires information erasure. Policy p requires information erasure when $\mathsf{reqErase}(p, \sigma)$ holds, where σ is the current memory. For example, policies $L \,{}^{x\geq 0}\!\nearrow H$ and $(L \,{}^{x=3}\!\nearrow H) \searrow^{y} L$ both require information erasure if $\sigma(x) = 3$. Since conditions are expressions, a condition may become satisfied when the memory is updated. The operational semantics for commands that update memory (assignment and declassification) use the utility function $\mathit{update}(\sigma, x, v)$ to overwrite variables, defined in Figure 6. The function $\mathit{update}(\sigma, x, v)$ takes memory σ, variable x, and integer v, and, provided policy $\Gamma(x)$ does not require erasure, returns $\mathit{erasure}(\sigma[x \mapsto v])$. The utility function $\mathit{erasure}(\sigma)$ checks for each variable y if policy $\Gamma(y)$ requires erasure given the memory σ; if so, it overwrites variable y with the value zero. Overwriting y changes the memory, and thus may trigger the overwriting of other variables.

The function $\mathit{erasure}(\sigma)$ is defined for all memories σ, and it provably overwrites variables as required: if $\sigma' = \mathit{erasure}(\sigma)$ then for all variables x, $\mathsf{reqErase}(\Gamma(x), \sigma')$ implies $\sigma'(x) = 0$.

Runtime mechanism for declassification. Declassification of information can occur only when appropriate conditions are satisfied. For example, policy $H \searrow^{x>0} L$ allows

OS-SKIP

$$\frac{}{\langle \textbf{skip}; c,\, \sigma\rangle \rightarrow \langle c,\, \sigma\rangle}$$

OS-ASSIGN

$$\frac{\sigma' = \mathit{update}(\sigma, x, \sigma(e))}{\langle x := e,\, \sigma\rangle \rightarrow \langle \textbf{skip},\, \sigma'\rangle}$$

OS-SEQUENCE

$$\frac{\langle c_0,\, \sigma\rangle \rightarrow \langle c_0',\, \sigma'\rangle}{\langle c_0; c_1,\, \sigma\rangle \rightarrow \langle c_0'; c_1,\, \sigma'\rangle}$$

OS-IF

$$\frac{i = \begin{cases} 0 & \text{if } \sigma(e) \neq 0 \\ 1 & \text{if } \sigma(e) = 0 \end{cases}}{\langle \textbf{if}\ e\ \textbf{then}\ c_0\ \textbf{else}\ c_1,\, \sigma\rangle \rightarrow \langle c_i,\, \sigma\rangle}$$

OS-WHILE

$$\frac{}{\langle \textbf{while}\ e\ \textbf{do}\ c,\, \sigma\rangle \rightarrow \langle \textbf{if}\ e\ \textbf{then}\ c; \textbf{while}\ e\ \textbf{do}\ c\ \textbf{else}\ \textbf{skip},\, \sigma\rangle}$$

OS-DECLASSIFY

$$\frac{v = \begin{cases} \sigma(e) & \text{if } \forall i \in 0..k.\ \sigma(e_i) \neq 0 \\ 0 & \text{if } \exists i \in 0..k.\ \sigma(e_i) = 0 \end{cases} \qquad \sigma' = \mathit{update}(\sigma, x, v)}{\langle x := \textbf{declassify}(e, p_f\ \textbf{to}\ p_t\ \textbf{using}\ e_0, \ldots, e_k),\, \sigma\rangle \rightarrow \langle \textbf{skip},\, \sigma'\rangle}$$

Figure 5: Operational semantics of IMP_E

information to be declassified to L when the expression $x > 0$ is non-zero, that is, when x is positive. The operational semantics for a guarded declassification command, x := **declassify**(e, p_f **to** p_t **using** $e_0, \ldots, e_k$), evaluates e and assigns the result to variable x provided the expressions $e_0, \ldots, e_k$ all evaluate to non-zero values. If one or more expressions e_i evaluate to zero, then declassification fails, and variable x is updated with the constant value zero. (Other reasonable semantics include leaving the value of x unchanged, or stopping execution.)

For example, if the policy $H \searrow^{\texttt{bar}>0} L$ is enforced on variable `foo`, then the command

$$\texttt{quux} := \textbf{declassify}(\texttt{foo}, H \searrow^{\texttt{bar}>0} L\ \textbf{to}\ L\ \textbf{using}\ \texttt{bar} > 0)$$

will successfully declassify the contents of `foo` only if the expression $\texttt{bar} > 0$ evaluates to a non-zero value.

The use of runtime mechanisms to aid in the enforcement of declassification and erasure policies allows simpler static enforcement mechanisms. The policies can be enforced without these runtime mechanisms, but would require either more complex static enforcement, or less expressive conditions. See Section 7 for more discussion on this tradeoff.

3.3 Type system

The runtime mechanisms of IMP_E ensure that declassification only occurs if appropriate conditions are satisfied, and that variables are overwritten when their policies require erasure. However, the runtime mechanisms alone are not sufficient to ensure that erasure and declassification policies are enforced. What prevents information with erasure policy $L \nearrow^{a} H$ from being stored in a variable x that has policy L enforced on it? Information in variable x has low security enforced on it, and is not necessarily overwritten when condition a is satisfied. Similarly, what prevents information with policy H from being stored in a variable with policy $H \searrow^{a} L$ enforced on it, and then subsequently (and incorrectly) being declassified?

The type system of IMP_E restricts information flow within a program, ensuring that appropriate policies are enforced on information at all times. The type system restricts both explicit flows, where information flows from direct assignments to variables, and implicit flows [4], where information flows via the program's control structure. The type system does not restrict timing or termination channels.

The typing judgment $pc, \Gamma \vdash c\ \mathsf{com}$ means that command c is well-typed under typing context Γ and program counter policy pc. The program counter policy is used to restrict implicit flows. It is an upper bound on the policies of information that may have influenced the value of the program counter, and so is an upper bound on the information that may be gained by knowing that command c is executed. The typing judgment $\Gamma \vdash e : p\ \mathsf{exp}$ means that under typing context Γ, policy p is an upper bound on the policies of information that may be gained by evaluating expression e.

Figure 7 presents inference rules for these typing judgments. The rules track and restrict the flow of information within a program. For example, the rule T-ASSIGN for an assignment $x := e$ ensures that information that may be revealed by evaluating expression e is allowed to flow to variable x ($\vdash p_e \leq \Gamma(x)$), and that information that may be revealed by learning the assignment is executed is also allowed to flow to variable x ($\vdash pc \leq \Gamma(x)$).

All the inference rules for the judgments $pc, \Gamma \vdash c\ \mathsf{com}$ and $\Gamma \vdash e : p\ \mathsf{exp}$ are standard for information-flow security type systems, with the exception of the rule for guarded declassification, T-DECLASSIFY. A guarded declassification command x := **declassify**(e, p_f **to** p_t **using** $e_0, \ldots, e_k$) declassifies information with policy p_f to policy p_t. Rule T-DECLASSIFY requires that p_f can be relabeled p_t assuming conditions conditions $e_0, \ldots, e_k$ are satisfied ($e_0, \ldots, e_k \vdash p_f \leq p_t$). Rule T-DECLASSIFY also requires that the declassified information is allowed to be stored in x ($\vdash p_t \leq \Gamma(x)$), that the information gained by knowing the declassification occurred can flow to x ($\vdash pc \leq \Gamma(x)$), and that the information gained by evaluating e is bounded above

T-SKIP
$$\frac{\Gamma \vdash pc\ \mathsf{pol}}{pc, \Gamma \vdash \textbf{skip}\ \mathsf{com}}$$

T-ASSIGN
$$\frac{\Gamma \vdash e \ :\ \Gamma(x)\ \mathsf{exp} \quad \vdash pc \leq \Gamma(x) \quad \Gamma \vdash pc\ \mathsf{pol}}{pc, \Gamma \vdash x := e\ \mathsf{com}}$$

T-SEQUENCE
$$\frac{pc, \Gamma \vdash c_0\ \mathsf{com} \quad pc, \Gamma \vdash c_1\ \mathsf{com}}{pc, \Gamma \vdash c_0; c_1\ \mathsf{com}}$$

T-IF
$$\frac{\Gamma \vdash e \ :\ p_e\ \mathsf{exp} \quad pc', \Gamma \vdash c_0\ \mathsf{com} \quad pc', \Gamma \vdash c_1\ \mathsf{com} \qquad \vdash pc \leq pc' \quad \vdash p_e \leq pc' \quad \Gamma \vdash pc\ \mathsf{pol}}{pc, \Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_0\ \textbf{else}\ c_1\ \mathsf{com}}$$

T-WHILE
$$\frac{\Gamma \vdash e \ :\ p_e\ \mathsf{exp} \quad pc', \Gamma \vdash c\ \mathsf{com} \qquad \Gamma \vdash pc\ \mathsf{pol} \quad \vdash pc \leq pc' \quad \vdash p_e \leq pc'}{pc, \Gamma \vdash \textbf{while}\ e\ \textbf{do}\ c\ \mathsf{com}}$$

T-DECLASSIFY
$$\frac{\Gamma \vdash e \ :\ p_f\ \mathsf{exp} \quad \vdash pc \leq \Gamma(x) \quad \vdash p_t \leq \Gamma(x) \quad \Gamma \vdash pc\ \mathsf{pol} \qquad \forall i \in 0..k.\ \Gamma \vdash e_i \ :\ \Gamma(x)\ \mathsf{exp} \quad e_0, \ldots, e_k \vdash p_f \leq p_t}{pc, \Gamma \vdash x := \textbf{declassify}(e, p_f\ \textbf{to}\ p_t\ \textbf{using}\ e_0, \ldots, e_k)\ \mathsf{com}}$$

T-VAL
$$\frac{}{\Gamma \vdash n \ :\ p\ \mathsf{exp}}$$

T-VAR
$$\frac{\vdash \Gamma(x) \leq p}{\Gamma \vdash x \ :\ p\ \mathsf{exp}}$$

T-OP
$$\frac{\Gamma \vdash e_0 \ :\ p_0\ \mathsf{exp} \quad \Gamma \vdash e_1 \ :\ p_1\ \mathsf{exp} \qquad \vdash p_0 \leq p \quad \vdash p_1 \leq p}{\Gamma \vdash e_0 \oplus e_1 \ :\ p\ \mathsf{exp}}$$

T-POL
$$\frac{\forall e \in \mathsf{eraseConds}(p).\ \Gamma \vdash e \ :\ p\ \mathsf{exp}}{\Gamma \vdash p\ \mathsf{pol}}$$

$$\mathsf{eraseConds}(\ell) \triangleq \emptyset$$
$$\mathsf{eraseConds}(p \searrow^{a} q) \triangleq \mathsf{eraseConds}(p)$$
$$\mathsf{eraseConds}(p \nearrow^{a} q) \triangleq \{a\} \cup \mathsf{eraseConds}(p)$$

Figure 7: Inference rules for typing judgments $pc, \Gamma \vdash c\ \mathsf{com}$, $\Gamma \vdash e \ :\ p\ \mathsf{exp}$, and $\Gamma \vdash p\ \mathsf{pol}$

by policy p_f ($\Gamma \vdash e \ :\ p_f\ \mathsf{exp}$).

There is a flow of information from the conditions $e_0, \ldots, e_k$ to the variable x. The operational semantics for a guarded declassification will assign the result of evaluating e into x only if all conditions $e_0, \ldots, e_k$ evaluate to non-zero values. Thus, the value of the variable x after the declassification command may reveal information about the value of the conditions. The typing rule for declassification, T-DECLASSIFY, tracks this information flow by requiring $\Gamma(x)$ to be an upper bound on the information that may be gained by knowing if condition e_i was satisfied ($\Gamma \vdash e_i : \Gamma(x)\ \mathsf{exp}$).

3.3.1 Well-formed contexts

A variable x is overwritten when $\Gamma(x)$, the policy enforced on x, requires erasure. Thus, if satisfaction of condition e can cause policy $\Gamma(x)$ to require erasure, there is information flow from e to x. To track and control this information flow, we restrict the typing contexts that may be used.

For all variables x, we require that policy $\Gamma(x)$ is well-typed, written $\Gamma \vdash \Gamma(x)\ \mathsf{pol}$. Any policy that is used as a program counter policy in the proof of a typing judgment $pc, \Gamma \vdash c\ \mathsf{com}$ must also be well-typed. The inference rule for well-typed policies is given in Figure 7. It requires that if condition e may cause policy p to require erasure, then p is an upper bound on the information that may be obtained by evaluating e ($\Gamma \vdash e \ :\ p\ \mathsf{exp}$).

The recursively defined function $\mathsf{eraseConds}(p)$ returns the set of expressions that may cause policy p to require erasure. That is, $\mathsf{reqErase}(p, \sigma)$ if and only if there is some condition $e \in \mathsf{eraseConds}(p)$ such that $\sigma(e) \neq 0$.

In addition, typing contexts are restricted to prevent infinite chains of variables $x_0, x_1, \ldots$, such that the overwriting of variable x_i depends on the value of variable x_{i+1}. For example, this restriction prevents a variable x having policy $L \nearrow^{x=0} H$. This restriction makes it easier to track information flows that occur due to overwriting, and simplifies both security proofs and implementation of variable overwriting. We define the *overwrite dependency* relation $\prec_\Gamma$ over variables such that $x \prec_\Gamma y$ if changing the value of x may cause policy $\Gamma(y)$ to require erasure. More formally, $x \prec_\Gamma y$ if there is an expression e such that $e \in \mathsf{eraseConds}(\Gamma(y))$ and x appears in e.

Definition 1 (Well-formed typing context) *Typing context* Γ *is* well-formed *if the overwrite dependency relation* $\prec_\Gamma$ *is well-founded and for all* $x \in$ *Vars,* $\Gamma \vdash \Gamma(x)\ \mathsf{pol}$.

3.4 Example

Figure 8 shows a fragment of IMP_E code that could be used to process a client request to the medical information website described in the introduction. For ease of presentation, we assume the existence of functions and strings.

The code first checks if the user has requested to exit the diagnosis application, and if so, sets variable `appEnd` and exits. Otherwise, the code gets the user's symptoms and uses them to produce a diagnosis, which would then be displayed to the user. Modulo the use of strings and functions, the code

```
1  if ( userReqExit ) then
2    appEnd = 1; exit()
3  else
4    // get user's symptoms
5    symp := getUserSymptoms();
6    ...
7    // diagnosis
8    if (contains(symp, 'malaise') &&
9        contains(symp, 'fever') && ...)
10     then diag := 'Influenza'
11     else if ...
```

$\Gamma(\texttt{symp}) = \mathit{session}\ {}^{\texttt{appEnd}}\!\!\nearrow \top \qquad \Gamma(\texttt{appEnd}) = \mathit{session}$

$\Gamma(\texttt{diag}) = \mathit{session}\ {}^{\texttt{appEnd}}\!\!\nearrow \top \qquad \Gamma(\texttt{userReqExit}) = \mathit{session}$

Figure 8: Medical information website example

is well-typed, and the relevant parts of the typing context Γ are also shown in Figure 8.

The policy enforced on the user symptoms, $\Gamma(\texttt{symp})$, is $\mathit{session}\ {}^{\texttt{appEnd}}\!\!\nearrow \top$. As described in Section 2, $\mathit{session}$ is a confidentiality level allowing only the session client and server to read the information, and $\top$ is a confidentiality level so restrictive that it prevents the server from storing the information. There is an implicit flow of information from symp to diag, as symp is used in the conditional test on lines 8–9, and diag is assigned to in the body of the conditional. By typing rule T-IF, the program counter policy for the conditional's body must be at least as restrictive as $\Gamma(\texttt{symp})$. Similarly, by rule T-ASSIGN, $\Gamma(\texttt{diag})$ must be as restrictive as that program counter policy. These constraints are satisfied by using policy $\Gamma(\texttt{symp})$ as the program counter policy for the body of the conditional, since $\Gamma(\texttt{symp}) = \Gamma(\texttt{diag})$.

The value of variable appEnd can cause policy $\mathit{session}\ {}^{\texttt{appEnd}}\!\!\nearrow \top$ to require erasure. Indeed, when variable appEnd is set (line 2), variables symp and diag are overwritten. There is thus information flow from appEnd to symp and diag. The requirement for a well-formed typing context tracks this flow, and requires that $\vdash \Gamma(\texttt{appEnd}) \leq \Gamma(\texttt{symp})$ and $\vdash \Gamma(\texttt{appEnd}) \leq \Gamma(\texttt{diag})$, which are satisfied, as

$$\Gamma(\texttt{appEnd}) = \mathit{session},$$
$$\Gamma(\texttt{symp}) = \Gamma(\texttt{diag}) = \mathit{session}\ {}^{\texttt{appEnd}}\!\!\nearrow \top,$$

and

$$\vdash \mathit{session} \leq \mathit{session}\ {}^{\texttt{appEnd}}\!\!\nearrow \top.$$

4 Security

The type system and runtime mechanisms of IMP_E correctly enforce the security policies of Section 2.

4.1 Noninterference

Noninterference [6] is a well-known end-to-end semantic security condition which requires that secret inputs do not influence public outputs. A formal statement of noninterference depends on the definitions of secret input and public output. In this paper, we consider the secret input to be the contents of a single variable at the start of program execution, and the public output to be the values of some subset of variables during execution. To state noninterference formally, we define notions of observational equivalence of configurations, execution traces, and correspondences between traces.

The *observation level* of variable x is determined by the policy $\Gamma(x)$ enforced on information stored in x. For policy p, $\mathsf{obs}(p) \in \mathcal{L}$ is the confidentiality level that is currently enforced on information labeled p, defined in Figure 9. The observation level of variable x is $\mathsf{obs}(\Gamma(x))$. Note that the observation level of a variable does not change during execution. The intuition is that a user with security clearance ℓ is only able to see the contents of variables with an observation level bounded above by ℓ. For example, if variable x has policy $(H \searrow^{a} L)\ {}^{b}\!\!\nearrow H$ enforced on it, the observation level of x is H, and a user with clearance L could not observe the contents of x. The policy $(H \searrow^{a} L)\ {}^{b}\!\!\nearrow H$ describes how the confidentiality of information stored in x may and must change as conditions are satisfied, but does not change the observability of the variable itself.

$$\mathsf{obs}(\ell) \triangleq \ell$$
$$\mathsf{obs}(p \searrow^{a} q) \triangleq \mathsf{obs}(p)$$
$$\mathsf{obs}(p\ {}^{a}\!\!\nearrow q) \triangleq \mathsf{obs}(p)$$

Figure 9: Observation level

Two configurations $\langle c, \sigma\rangle$ and $\langle c', \sigma'\rangle$ are *observationally equivalent at level* ℓ, written $\langle c, \sigma\rangle \approx_\ell \langle c', \sigma'\rangle$, if all variables that are observable at level ℓ have the same value in both memories. Observational equivalence is implicitly parameterized on the typing context Γ. More formally, $\langle c, \sigma\rangle \approx_\ell \langle c', \sigma'\rangle$ if and only if for all $x \in \mathit{Vars}$, if $\mathsf{obs}(\Gamma(x)) \sqsubseteq \ell$ then $\sigma(x) = \sigma'(x)$. Intuitively, if $\langle c, \sigma\rangle \approx_\ell \langle c', \sigma'\rangle$, then a user with security clearance ℓ is unable to distinguish these two configurations by examining the contents of the memory. However, a user may be able to distinguish two executions of the program starting from $\langle c, \sigma\rangle$ and $\langle c', \sigma'\rangle$, by observing the sequences of configurations that each execution produces. This motivates the definition of traces, and correspondences between traces.

A *trace* τ is a (finite or infinite) sequence of configurations $\tau = \langle c_0, \sigma_0\rangle\langle c_1, \sigma_1\rangle \dots$ such that $\langle c_{i-1}, \sigma_{i-1}\rangle \rightarrow \langle c_i, \sigma_i\rangle$ for all $i \in \mathbb{N}$ such that $0 < i < |\tau|$, where $|\tau|$ denotes the length of trace τ. We write $\tau[i]$ to refer to the ith configuration in the trace τ.

We use *correspondences* [1] between traces to indicate which states appear equivalent to an observer that sees first one trace, then the other. A correspondence R is a relation over the natural numbers. If R is a correspondence for traces τ_1 and τ_2, and $(i, j) \in R$, we will use it to mean that $\tau_1[i]$ and $\tau_2[j]$ look the same to a given observer. Formally, a correspondence R between traces τ_1 and τ_2 is a subset of $\mathbb{N} \times \mathbb{N}$

such that

1. (Completeness) either $\{i \mid (i,j) \in R\} = \{i \in \mathbb{N} \mid i < |\tau|\}$ or $\{j \mid (i,j) \in R\} = \{j \in \mathbb{N} \mid j < |\tau'|\}$; and
2. (Initial configurations) if $|R| > 0$ then $(0,0) \in R$; and
3. (Monotonicity) for all $(i,j) \in R$ and $(i',j') \in R$, if $i < i'$ then $j \leq j'$; and, symmetrically, if $j < j'$ then $i \leq i'$.

This definition ensures that a correspondence covers all configurations in at least one of τ or τ', and if both traces are non-empty, then the initial configurations in the traces correspond to each other. The monotonicity requirement implies that the observer observes each trace as it executes, and time moves only forward.

Correspondences are both timing and termination insensitive, implicitly assuming that an observer cannot directly observe atomic transitions, and cannot detect if an execution has terminated. The definition can be refined to provide timing and/or termination sensitivity. Termination sensitivity is achieved by strengthening completeness to require that the correspondence covers all configurations in both τ and τ', and that no configuration in τ or τ' corresponds to an infinite set of configurations. Timing sensitivity is achieved by strengthening the definition so that every configuration in τ and τ' corresponds to exactly one other configuration. Timing sensitivity implies an observer is able to observe each time step, and entails termination sensitivity.

Having defined traces, correspondences, and observational equivalence of configurations, we can now state noninterference. A command is noninterfering at level ℓ for variable x, if input supplied in the variable x at the beginning of the program has no observable effect for a user with security clearance ℓ, watching the execution of the system:

Definition 2 (Noninterference) *A command c with typing context Γ is* noninterfering at level ℓ for variable x *if for all integers $v_1, v_2 \in \mathbb{Z}$, all memories σ, and all traces τ_1 and τ_2 such that $\tau_i[0] = \langle c, \mathit{update}(\sigma, x, v_i)\rangle$ for $i \in \{1,2\}$, there exists a correspondence R for τ_1 and τ_2 such that for all $(i,j) \in R$, $\tau_1[i] \approx_\ell \tau_2[j]$.*

The definition of noninterference relies on a typing context Γ, used in the definition of observational equivalence. For brevity, we omit mention of Γ when clear from context.

Noninterference is too strong in the presence of declassification, which intentionally makes secret information public. Noninterference cannot express erasure requirements, which make publicly observable information less observable. Motivated by these shortcomings of noninterference, we defined noninterference according to policy.

4.2 Noninterference according to policy

Noninterference according to policy [2] is a semantic security condition that generalizes noninterference, and allows precise reasoning about the observability of information as it undergoes declassification and erasure.

Noninterference according to policy is defined in terms of the policy semantics, presented in Section 2. The intuition behind the policy semantics is that if information in state s has policy p enforced on it, then when the system enters state s', the information (or anything derived or influenced by it) should be observable at level ℓ only if $(s', \ell) \in [\![p]\!]_s$. Noninterference according to policy makes this intuition precise. Here, we specialize the definition of noninterference according to policy for IMP_E programs.

Definition 3 (Noninterference according to policy) *A command c with typing context Γ is* noninterfering according to policy for variable x *if for all integers $v_1, v_2 \in \mathbb{Z}$, all memories σ, memories $\sigma_1 = \mathit{update}(\sigma, x, v_1)$ and $\sigma_2 = \mathit{update}(\sigma, x, v_2)$, and all traces τ_1 and τ_2 such that $\tau_i[0] = \langle c, \sigma_i\rangle$ for $i \in \{1,2\}$, there exists a correspondence R for τ_1 and τ_2 such that for all $(i,j) \in R$, for all $\ell \in \mathcal{L}$, if $(\tau_1[i], \ell) \notin [\![\Gamma(x)]\!]_{\langle c, \sigma_1\rangle}$ and $(\tau_2[j], \ell) \notin [\![\Gamma(x)]\!]_{\langle c, \sigma_2\rangle}$, then $\tau_1[i] \approx_\ell \tau_2[j]$.*

Like noninterference, noninterference according to policy places restrictions on whether information input in variable x is observable by a user during the execution of the program. However, whereas noninterference required all corresponding configurations to be observationally equivalent at a fixed level ℓ, noninterference according to policy is more precise, and requires corresponding configurations to be observationally equivalent at confidentiality levels determined by the semantics of the policy enforced on the input. Thus, noninterference according to policy reflects how the observability of input may change during the execution of the system, as declassifications and erasures occur.

Noninterference according to policy generalizes noninterference. In particular, if the policy enforced on a variable x indicates that information will never be observable at a confidentiality level ℓ, then noninterference according to policy for variable x implies noninterference at level ℓ for variable x. For example, a program that is noninterfering according to policy and takes input in variable x with policy H enforced on it, will never declassify the input to level L, and thus is noninterfering at level L for x. The following theorem states this formally.

Theorem 1 *For all commands c, typing contexts Γ, and variables x, if c is noninterfering according to policy for variable x, then for all confidentiality levels ℓ such that for all memories σ, $\ell \notin \{\ell' \mid (s, \ell') \in [\![\Gamma(x)]\!]_{\langle c, \sigma\rangle}\}$, c is noninterfering at level ℓ for variable x.*

The central result of this paper is that the type system and runtime mechanisms of IMP_E suffice to enforce erasure and declassification policies. Thus, any well-typed IMP_E program is noninterfering according to policy.

Theorem 2 *For all typing contexts Γ and commands c, if Γ is well-formed, and $pc, \Gamma \vdash c\ \mathsf{com}$ for some policy pc, then for all variables $x \in \mathit{Vars}$, c is noninterfering according to policy for variable x.*

The proof of Theorem 2 is given in Appendix B. It uses Pottier and Simonet's noninterference proof technique [19].

5 To Jif and beyond

The Jif programming language [17] extends Java [7] with information-flow control, allowing security policy annotations on program variables and method signatures. In this section, we describe how we extended Jif with declassification and erasure policies, and mechanisms to enforce these policies. The resulting language is called Jif_E.

5.1 Decentralized label model

Security policies in Jif are from the *decentralized label model* (DLM) [16]. In DLM labels, security principals declare confidentiality and integrity restrictions on information. The *reader policy* $o \rightarrow r$ means that the principal o owns the policy, and allows principal r to learn, or read, information; the *writer policy* $o \leftarrow w$ is also owned by principal o, who allows principal w to influence, or write, information.[1] A *label* consists of conjunctions ($\sqcup$) and disjunctions ($\sqcap$) of reader and writer policies. Within a label, different principals may declare different restrictions, making the DLM suitable for reasoning about security in the presence of mutual distrust between principals. Variable types and method signatures in Jif may be annotated with labels. A *labeled type* is a pair of a base type (a primitive type or class) and a label.

We extended the DLM to allow principals to specify confidentiality restrictions using declassification and erasure policies. That is, declassification and erasure policies may now appear in reader policies on the right of the arrow.

The base lattice of confidentiality levels is the set of security principals, which is closed under conjunction ($\wedge$) and disjunction ($\vee$) [13, 23], and so forms the necessary lattice structure. For example, the reader policy $Alice \rightarrow (Bob \vee Chuck) \overset{a}{\nearrow} Bob$ is owned by Alice, who requires the erasure policy $(Bob \vee Chuck) \overset{a}{\nearrow} Bob$ to be enforced. The erasure policy initially allows Bob or Chuck to read information, but once a is satisfied, only Bob may read it.

Instead of security principals, we could have used the decentralized labels as the base lattice. This would allow labels such as $(Alice \rightarrow Bob) \overset{a}{\searrow} (Chuck \rightarrow Dave)$. However, this approach runs counter to the philosophy of decentralization, because it prevents different principals from declaring their own declassification and erasure requirements.

For the condition language, we allow a restricted class of expressions: *access path expressions* of type `condition`, and negations of these access path expressions. The type `condition` is a new primitive type with two values: `true` and `false`. Expressions of type `condition` may be cast to `boolean`, and vice versa. An access path expression is an expression of the form $r.f_1.\ldots.f_n$, where r is a local variable, the special variable `this`, or a class name; each f_i is a field; and all path elements other than the last are declared `final`. Immutability of path elements is needed for sound reasoning about conditions within the type system.

[1] The mnemonic for arrow direction in reader and writer policies is that in a reader policy $o \rightarrow r$, information may flow *to* principal r, whereas in a writer policy $o \leftarrow w$, information may flow *from* principal w.

5.2 Syntax and semantics

Jif_E extends Jif's syntax and runtime system to incorporate the guarded declassification syntax and runtime erasure mechanisms of Section 3.

Jif_E contains the new guarded declassification expression **declassify**$(e, L_f$ **to** L_t **using** $e_0, \ldots, e_k)$, where L_f and L_t are labels, and each expression e_i is of type `condition`. The expression is evaluated by first evaluating e to a value v, then evaluating each e_i in turn; if any e_i evaluates to `false`, then an `UnsatisfiedConditionException` is thrown; otherwise, the expression evaluates to v. If the evaluation of e or any e_i results in an exception, the declassification expression also results in the exception. As in the typing rule for declassification in Figure 7, type checking ensures that L_f may be relabeled L_t under the assumption that all conditions e_i are satisfied.

Note that Jif already provides a mechanism for *selective declassification* [15, 14, 18], whereby a declassification that weakens or removes a policy owned by principal o requires o's authority. By contrast, guarded declassification does not require the authority of any principal, since given a reader policy $o \rightarrow (p \overset{a}{\searrow} q)$, the principal o has already stated that information may be declassified when condition a is satisfied. In Jif_E, selective declassification and guarded declassification coexist as separate and independent mechanisms.

To enforce erasure policies, Jif_E ensures that a variable or location that has label L enforced on it is overwritten whenever any erasure policy in L requires it. For example, if a location has the label $\{Alice \rightarrow (Bob \overset{\texttt{this.f}}{\nearrow} Chuck) \sqcap Dave \rightarrow (Alice \overset{\texttt{this.o.d}}{\nearrow} \top)\}$ enforced on it, then the location is overwritten whenever either `this.f` or `this.o.d` evaluates to `true`. When a location or variable is overwritten, its contents are replaced with an appropriate default value. Thus, numeric locations are overwritten with zero, and reference locations are overwritten with `null`. Section 5.4 describes the runtime mechanisms used to achieve this. This erasure mechanism is analogous to the erasure mechanism of IMP_E, which overwrites variables if the policy enforced on the variable requires erasure.

5.2.1 Interaction with Java and Jif features

Jif is intended for practical information-flow control. It supports a large subset of Java's language features, and provides additional features such as dynamic labels, constant arrays, and class and method polymorphism, needed for building real applications. The erasure enforcement mechanism of IMP_E needs careful adaptation for these language features.

Final fields and variables. In Java, fields, local variables, and formal arguments can be marked `final`, meaning their value will not change after initialization. To respect the finality of variables and locations, Jif_E requires that final variables and fields cannot be overwritten. The label L enforced on a final field or variable must not contain any erasure policies, and if L contains a dynamic label (see below), then the dynamic label must not contain any erasure policies. This ensures that label L never requires erasure.

Arrays. Jif allows different labels to be enforced on the elements of an array and the array itself. If the label enforced on the elements of an array requires erasure, the array is overwritten with appropriate default values; the length of the array is not altered. Jif supports *constant arrays*, whose elements cannot be modified after initialization. As with `final` fields, labels on elements of constant arrays must never require erasure.

Dynamic labels. Jif can represent labels at runtime and treat labels as first-class values. The primitive type `label` is the type of runtime labels, and Jif permits runtime comparisons of dynamic labels. Jif$_E$ extends the runtime representation of labels to permit declassification and erasure policies to also be represented at runtime. We introduce a new kind of label, to reason about runtime labels that may require erasure. The primitive type `elabel` is used for dynamic labels that may require erasure. Only dynamic labels of type `elabel` may contain erasure policies; a dynamic label of type `label` cannot contain erasure policies. Thus, the labels of final fields, final variables, and elements of constant arrays, may refer to dynamic labels of type `label`, but may not refer to dynamic labels of type `elabel`. The type `label` can be cast to `elabel`, but not vice versa. The restriction that only `elabels` may contain erasure policies also simplifies backwards compatibility of Jif$_E$ with Jif.

Polymorphism. Jif provides polymorphism for the labels of method arguments. For example, the method signature `double`$\{a\}$ `sine`(`double`$\{Alice \rightarrow Bob\}$ `a`) states that the label on the value returned is the same as the label of the actual argument `a`, which can be no more restrictive than $\{Alice \rightarrow Bob\}$. In Jif method bodies, the label of a formal argument is a polymorphic label, representing the label of actual argument, and bounded above by the argument label specified in the signature. However, because actual arguments may require erasure during the method body execution, we need to know what label to enforce on formal arguments in the method body. Thus, in Jif$_E$, method bodies assume that the label of a formal argument is simply the argument label bound specified in the signature. This is sound, but not as permissive as Jif, and effectively removes argument label polymorphism. However, it is not overly restrictive: we successfully implemented a remote voting system in 14,000 lines of Jif$_E$ code, as discussed in Section 6.

Jif also supports polymorphic classes, permitting classes to be parameterized on labels and principals.[2] Jif$_E$ extends the class parameters to allow parameters of type `elabel`.

[2]Jif as of version 3.1 does not support Java generics, another form of class parameterization for polymorphism.

5.3 Information flow

Jif's existing type-system tracks information flow. As discussed in Section 3, condition satisfaction can itself be used as a covert storage channel. Jif$_E$ extends Jif's type system to soundly track this potential information flow.

Condition satisfaction affects whether the expression **declassify**$(e, L_f$ **to** L_t **using** $e_0, \ldots, e_k)$ declassifies e or throws an `UnsatisfiedConditionException`. Jif$_E$ requires that the label of each e_i is no more restrictive than label L_t.

Condition satisfaction may also cause variables and locations to be overwritten. Jif$_E$ tracks these information flows analogously to the IMP$_E$ policy typing judgment $\Gamma \vdash p$ pol. Jif$_E$ requires that whenever a label L is declared in a program, for any erasure policy $p \overset{e}{\nearrow} q$ that occurring in L, the label of expression e must be no more restrictive than L. Jif$_E$ also requires that if `lbl` is a dynamic label that occurs in L, then the value `lbl` must be no more restrictive than L. So, if e is a condition that appears in `lbl`, then the label of e is no more restrictive than `lbl`, and thus no more restrictive than L.

5.4 Translation

The Jif compiler [17] is a source-to-source compiler, producing Java code as output. Jif programs rely on a small trusted runtime library, implemented in Java, that provides functionality such as runtime comparisons of labels. We extended the runtime library, and modified the source-to-source translation, to provide runtime support for erasure.

The key idea is that if a variable or location may need to be overwritten depending on the satisfaction of a condition a, then a listener is registered with condition a; the listener is notified whenever the value of a changes, and the listener will overwrite the variable or location if necessary.

If a local variable may need to be overwritten, then the translation moves the local variable to the heap, to allow a condition listener to access it, and overwrite it as needed.

Assignments to fields and local variables are translated to check that the variable or field does not currently require erasure. The combination of condition listeners and assignment checks ensures that whenever the label enforced on the variable or location requires erasure, the variable or location will be zero or `null` as appropriate.

Overwriting a variable or location of type `condition` may trigger the overwriting of other variables and locations. To ensure that updating a condition does not cause an infinite cascade of listener invocations, the type system of Jif$_E$ requires that for all conditions a, the value of a cannot (directly or indirectly) control whether a needs to be overwritten. This is analogous to ensuring that the overwrite dependency relation $\prec_\Gamma$ of Section 3 is well-founded.

6 Case study: Civitas

Using Jif$_E$, we implemented Civitas [3], a practical, secure, remote voting system. The use of declassification and erasure policies in the implementation of Civitas help ensure that the system's security requirements are satisfied. This section discusses the experience of using Jif$_E$ to implement Civitas.

Civitas guarantees strong security properties in the presence of a strong adversary. The design of Civitas refines a cryptographic voting scheme by Juels, Catalano, and Jakobsson [12]. The entities involved in a Civitas election include an election supervisor, voters, and *election authorities*, which are mutually distrusting entities that collaborate to run an election. A Civitas election has several phases.

1. *Setup.* The electoral roll is established and shared keys are generated.
2. *Registration.* Voters retrieve credentials from election authorities.
3. *Voting.* Voters vote using their credentials.
4. *Tabulation.* Election authorities tabulate the election results.

More details of the design and security assurances of Civitas are available in a recent publication [3].

Civitas is implemented in 14,000 lines of Jif$_E$ code, with about 8,000 additional lines of Java code to perform I/O and implement cryptographic operations. Declassification and erasure policies are used in four distinct places.

- *Generation of a shared key by authorities.* During setup, authorities engage in a protocol to generate a shared El Gamal key pair. Each authority generates a share of the key pair, and publishes a commitment to it. Each authority publishes its share of the public key, but only after all commitments are published.
  The label $\{A_i \rightarrow A_i \searrow^{allCommPosted} \bot; A_i \leftarrow A_i\}$ is used for authority A_i's public key share. The declassification policy requires that initially the information is readable only by election authority A_i, and may be declassified to be readable by everyone (represented by the bottom principal $\bot$) when condition $allCommPosted$ is satisfied. Condition $allCommPosted$ is a field of type `condition`. It is easy to check that this field is only updated once A_i has successfully retrieved all key commitments. The writer policy $A_i \leftarrow A_i$ indicates that the key share was influenced only by A_i.
- *Commit-reveal protocol by authorities.* During tabulation, the authorities jointly generate random bits, and each authority must believe that the bits are random. Each authority selects random bits, and publishes a commitment to these bits. Once all commitments are published, each authority reveals its bits, which can be combined to form a sequence of bits that all authorities agree are random.
  Similar to the key shares, the label $\{A_i \rightarrow A_i \searrow^{allBitsPosted} \bot; A_i \leftarrow A_i\}$ is used for authority A_i's random bits. Condition $allBitsPosted$ is a field of type `condition`, and it is easy to check that this field is only updated once A_i has been able to successfully retrieve all bit commitments.
- *Management of credential shares by authorities.* During registration, each authority generates a credential share for each voter. Each voter contacts each authority to retrieve his shares, combining them into a credential that can be used to vote. After delivering the share to the voter, the authority removes the share from the system. This helps ensure that the voter's anonymity is not violated should A_i be subsequently compromised.
  Authority A_i enforces the label $\{A_i \rightarrow (A_i \,{}^{delivered}\!\nearrow \top) \searrow^{deliveryReq} \bot; A_i \leftarrow A_i\}$ on each voter credential share. Condition $deliveryReq$ is satisfied when the voter has requested his credential share, and has authenticated himself to the authority. The satisfaction of this condition allows the declassification of the share.[3] Any copies of the information that were not declassified must be erased when condition $delivered$ is satisfied upon successful retrieval by the voter.
- *Management of voter credential shares by voting clients.* After voter V_j has retrieved all credential shares from the authorities, he combines them into a single credential, which he then uses to vote, publishing it together with his ballot. After combining the shares, the voter deletes them, to remove any record of which authority provided which share.
  The voter enforces on each credential share the label $\{V_j \rightarrow (V_j \,{}^{postCombined}\!\nearrow \top) \searrow^{combined} \bot\}$. Upon combining the shares into a credential, condition $combined$ is satisfied, and the voter can declassify the credential to allow it to be published with his ballot. After combining shares, condition $postCombined$ is satisfied, and undeclassified copies of the shares (or of information derived from them) are erased.

Jif$_E$ allows complex declassification and erasure security requirements to be clearly and unambiguously declared on the data. In addition to stating what security must currently be enforced on the data, the policies limit how the data may be used in the future. The information flow analysis ensures that uses of the data conform to the declared security policies. This provides additional assurance that the Civitas implementation is correct. The policy annotations serve as a form of documentation, making complex information security requirements visible in the code itself.

7 Related work

The most closely related work is that of Hunt and Sands [11]. Concurrently with this work, they consider the enforcement of simple erasure policies of the form $\ell \nearrow \ell'$, where erasure is required at the end of a lexical scope. These policies are

[3] Ideally, the declassification policy should allow the share to be readable only by the voter V_j it is intended for. In the protocol between authority A_i and V_j, each authenticates to the other, and they establish a shared key k; the credential share is sent to V_j encrypted with k. The reasoning supported by the DLM is not powerful enough to determine that information encrypted with k is readable only by A_i and V_j. Extending it to reason about the subtleties of cryptography would allow a more precise declassification policy, but is largely orthogonal to this work.

a restricted instantiation of the policy framework used here, where policies cannot be nested and the condition language is limited to specifying the end of lexical scopes. Using flow-sensitive typing contexts [10], Hunt and Sands present an elegant type system to enforce erasure policies; their system requires no runtime erasure mechanism.

Comparing our work to Hunt and Sands' highlights a tension between expressiveness of erasure conditions and ease of enforcement. Simpler condition languages are easier to reason about statically, and thus easier to enforce statically. Hunt and Sands' conditions are tied to lexical scopes, and it is straightforward to reason statically about when conditions are satisfied. By contrast, the condition language used in this work is program expressions: flexible, but difficult to reason about statically. Because it is difficult or impossible to know the value of an arbitrary expression at a given program point prior to execution, it is difficult to determine statically whether a policy will require erasure at that program point, and thus difficult to enforce erasure statically. Instead, we use a simple runtime mechanism to enforce erasure, an approach similar in spirit to hybrid type checking [5].

Although runtime mechanisms are used to help enforce erasure and declassification, information flow control in IMP_E is static, using a type system to track and restrict the flow of information. Starting with Volpano, Smith and Irvine [24], type systems have proven successful in providing information flow control without the overhead of representing security labels at runtime; many of these type systems are surveyed by Sabelfeld and Myers [20], and Sabelfeld and Sands [21] discuss some of the recent type systems that consider declassification.

Hansen and Probst [9] consider information flow security in Java Card bytecode, and identify the utility of erasure policies in providing security assurance. They consider "simple erasure policies" of the form $L \overset{end}{\nearrow} H$, where end is a condition indicating the end of execution of the current program. They define a corresponding *simple erasure* security condition. They conjecture, but do not demonstrate, that simple erasure is straightforward to enforce.

Hansen and Probst [8] have also used erasure policies in secure dynamic program repartitioning. Secure program partitioning [25] is a technique to split data and code across a set of mutually distrusting hosts while guaranteeing security. Hansen and Probst consider repartitioning a program when the set of hosts changes dynamically, and use erasure policies to ensure that old copies of data are removed from the system when repartitioning occurs. Hansen and Probst do not describe how to enforce the erasure policies. Søndergaard's subsequent master's thesis [22] discusses the trusted runtime components required to enforce these erasure policies, but does not implement them.

Sabelfeld and Sands [21] consider different aspects of declassification, and propose four semantic principles for declassification, three of which are applicable to erasure. Noninterference according to policy satisfies *semantic consistency* and *conservativity*, but does not satisfy *non-occlusion* precisely because, as Hunt and Sands [11] point out, the policies address *when*, but not *what*, information is erased and declassified.

8 Conclusion

In this paper we have shown how to enforce erasure requirements end-to-end in language-based settings. Erasure requirements are specified in a flexible and powerful policy framework [2] that can also express declassification requirements. The policies express when information may be declassified, and when information must be erased.

We have proved that an information-flow control type system, in conjunction with a runtime mechanism for erasure, can enforce the erasure and declassification policies in IMP_E, a simple imperative language. Well-typed IMP_E programs satisfy *noninterference according to policy* [2].

The end-to-end enforcement of erasure and declassification policies is also practical: we have extended the Jif programming language [17] with erasure and declassification policies and enforcement mechanisms, and used the resulting language to implement a secure remote voting system.

The ultimate goal of this work is to make it easy for programmers to write secure programs, and to have assurance that these programs are secure. This work, by providing provably secure enforcement of expressive erasure and declassification policies, brings us closer to that goal.

Acknowledgments

We thank Michael Clarkson for comments on an earlier draft, Fred Schneider for discussions on how best to present this material, and Eric Breck for suggesting the medical diagnosis application. We also thank the anonymous reviewers for their insightful comments.

References

[1] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. Technical Report CS Report 2007-04, Stevens Institute of Technology, Nov. 2007.

[2] S. Chong and A. C. Myers. Language-based information erasure. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005.

[3] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. In *Proc. IEEE Symposium on Security and Privacy*, May 2008.

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

[5] C. Flanagan. Hybrid type checking. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 245–256, 2006.

[6] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. ISBN 0-201-31008-2.

[8] R. R. Hansen and C. W. Probst. Secure dynamic program repartitioning. In *Proc. Nordic Workshop on Secure IT-Systems*, Oct. 2005.

[9] R. R. Hansen and C. W. Probst. Non-interference and erasure policies for Java Card bytecode. In *Proc. 6th International Workshop on Issues in the Theory of Security*, Mar. 2006.

[10] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 79–90, 2006.

[11] S. Hunt and D. Sands. Just forget it—the semantics and enforcement of information erasure. In *Proc. 17th European Symposium on Programming*, Mar. 2008.

[12] A. Juels, D. Catalano, and M. Jakobsson. Coercion-resistant electronic elections. In *Workshop on Privacy in the Electronic Society*, pages 61–70, Nov. 2005.

[13] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, October 1991. *Operating System Review*, 253(5).

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

[15] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, 1997.

[16] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000.

[17] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2001.

[18] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, 2000.

[19] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

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

[21] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[22] D. Søndergaard. Secure program partitioning in dynamic networks. Master's thesis, Technical University of Denmark, 2006. IMM-M.Sc-2006-92.

[23] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. 14th European Symposium on Programming*, 2005.

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

[25] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, pages 1–14, Oct. 2001.

A Sound relabeling judgment

Figure 10 shows inference rules for the $a_0, \ldots, a_k \vdash p \leq q$ judgment. The rule RL-LATTICE states that information may be relabeled from lattice policy ℓ to lattice policy ℓ' in any state, provided that $\ell \sqsubseteq \ell'$. The rule RL-TRANS makes the judgment transitive on policies.

The declassification rule RL-DECL permits relabeling from a declassification policy $p \overset{a}{\searrow} p'$ to policy p', provided that condition a is satisfied. This rule captures the intuitive meaning of declassification policies: declassification may occur when the appropriate condition is satisfied. Note that rule RL-DECL permits relabeling from $p \overset{a}{\searrow} p'$ to p', and p' may permit declassifications or require erasures that $p \overset{a}{\searrow} p'$ does not.

The declassification introduction rule RL-DECL-I describes when it is permissible to relabel information from some policy q to the policy $p \overset{a}{\searrow} p'$. First, it must be permitted to relabel information from q to p when $a_0, \ldots, a_k$ are satisfied; second, in any state where condition a is satisfied, it must be permitted to relabel information from q to p'.

The declassification elimination rule RL-DECL-E allows information to be relabeled from a declassification policy $p \overset{a}{\searrow} p'$ to the policy p. Intuitively, it is acceptable to relabel information from $p \overset{a}{\searrow} p'$ to p, since policy p is always more restrictive than policy $p \overset{a}{\searrow} p'$, which enforces everything that p does but also permits declassification to p'.

The rule RL-DECL-DECL describes when information may be relabeled from one declassification policy $p \overset{a}{\searrow} p'$ to another, more restrictive declassification policy $q \overset{b}{\searrow} q'$. The intuition is that this is permitted if q is at least as restrictive as p when $a_0, \ldots, a_k$ are satisfied, the policy $q \overset{b}{\searrow} q'$ permits declassification only when $p \overset{a}{\searrow} p'$ does (that is, $b = a$), and, whenever declassification is permitted, q' is at least as restrictive as p'.

As can be seen by inspection of Figure 10, each of the relabeling rules for erasure policies corresponds to a relabeling rule for declassification. For example, erasure introduction RL-ERASE-I is analogous to RL-DECL-E: information may be relabeled from p to $p \overset{a}{\nearrow} p'$, since $p \overset{a}{\nearrow} p'$ is always more restrictive than p. An erasure policy $p \overset{a}{\nearrow} p'$ enforces everything that p does, and in addition requires the information to be erased at certain times.

Erasure elimination RL-ERASE-E is analogous to the rule for declassification introduction, allowing information to be relabeled from $p \overset{a}{\nearrow} p'$ to q provided that p can be relabeled to q when conditions $a_0, \ldots, a_k$ are satisfied, and p' can be relabeled to q at all times. Intuitively, information may be relabeled to q since information labeled q would not need to be erased when a is satisfied, as q is at least as restrictive as both p and p'.

The rule RL-ERASE-ERASE compares two erasure policies, $p \overset{a}{\nearrow} p'$ and $q \overset{b}{\nearrow} q'$, and is similar to RL-DECL-DECL. Information may be relabeled from $p \overset{a}{\nearrow} p'$ to $q \overset{b}{\nearrow} q'$ provided that q is at least as restrictive as p when $a_0, \ldots, a_k$ are satisfied, and whenever $p \overset{a}{\nearrow} p'$ requires information to be erased, so does $q \overset{b}{\nearrow} q'$ (that is, $b = a$), and at all times, q' is at least as restrictive as p'.

There is no erasure rule analogous to RL-DECL. This is because erasure policies specify information flows that must not happen, which is difficult to capture with inference rules of this style. Instead, the onus of ensuring information is

$$\text{RL-LATTICE}\quad \frac{\ell \sqsubseteq \ell'}{a_0, \ldots, a_k \vdash \ell \le \ell'} \qquad \text{RL-TRANS}\quad \frac{\begin{array}{c} a_0, \ldots, a_k \vdash p \le p' \\ a_0, \ldots, a_k \vdash p' \le p'' \end{array}}{a_0, \ldots, a_k \vdash p \le p''} \qquad \text{RL-DECL}\quad \frac{a \in \{a_0, \ldots, a_k\}}{a_0, \ldots, a_k \vdash p \overset{a}{\searrow} p' \le p'}$$

$$\text{RL-DECL-I}\quad \frac{\begin{array}{c} a_0, \ldots, a_k \vdash q \le p \\ a \vdash q \le p' \end{array}}{a_0, \ldots, a_k \vdash q \le p \overset{a}{\searrow} p'} \qquad \text{RL-DECL-E}\quad \frac{}{a_0, \ldots, a_k \vdash p \overset{a}{\searrow} p' \le p} \qquad \text{RL-DECL-DECL}\quad \frac{\begin{array}{c} a_0, \ldots, a_k \vdash p \le q \\ a \vdash p' \le q' \end{array}}{a_0, \ldots, a_k \vdash p \overset{a}{\searrow} p' \le q \overset{a}{\searrow} q'}$$

$$\text{RL-ERASE-I}\quad \frac{}{a_0, \ldots, a_k \vdash p \le p \overset{a}{\nearrow} p'} \qquad \text{RL-ERASE-E}\quad \frac{\begin{array}{c} a_0, \ldots, a_k \vdash p \le q \\ \vdash p' \le q \end{array}}{a_0, \ldots, a_k \vdash p \overset{a}{\nearrow} p' \le q} \qquad \text{RL-ERASE-ERASE}\quad \frac{\begin{array}{c} a_0, \ldots, a_k \vdash p \le q \\ \vdash p' \le q' \end{array}}{a_0, \ldots, a_k \vdash p \overset{a}{\nearrow} p' \le q \overset{a}{\nearrow} q')}$$

Figure 10: Inference rules for $a_0, \ldots, a_k \vdash p \le q$

erased at appropriate times falls upon the system that enforces the policies.

Theorem 3 *If $a_0, \ldots, a_k \vdash p \le q$ then for all states s, such that $\forall i \in 0..k.\ s \vDash a_i$, we have $[\![p]\!]_s \supseteq [\![q]\!]_s$.*

Proof: We proceed by induction on the proof of $a_0, \ldots, a_k \vdash p \le q$. The inductive hypothesis is that for any premise of the form $a_0, \ldots, a_k \vdash p' \le q'$, we have $[\![p']\!]_s \supseteq [\![q']\!]_s$ for any state s such that $\forall i \in 0..k.\ s \vDash a_i$.

- **RL-LATTICE, RL-TRANS.** Trivial.
- **RL-DECL.** Here $p \equiv p' \overset{a}{\searrow} q$, and $s \vDash a$. We have $[\![p' \overset{a}{\searrow} q]\!]_s = [\![p']\!]_s \cup \bigcup\{[\![q]\!]_{s'} \mid s \longrightarrow^* s' \text{ and } s' \vDash a\} \supseteq [\![q]\!]_s$, since $s \vDash a$.
- **RL-DECL-I.** Here $q \equiv q' \overset{b}{\searrow} q''$, and $a_0, \ldots, a_k \vdash p \le q'$, and $b \vdash p \le q''$. By the inductive hypothesis, we have $[\![p]\!]_s \supseteq [\![q']\!]_s$ and $[\![p]\!]_{s'} \supseteq [\![q'']\!]_{s'}$ for any state s' such that $s' \vDash b$. Thus, $[\![p]\!]_s \supseteq [\![q]\!]_s$ as required.
- **RL-DECL-E.** Here $p \equiv q \overset{a}{\searrow} p'$, and $[\![q \overset{a}{\searrow} p']\!]_s = [\![q]\!]_s \cup \bigcup\{[\![p']\!]_{s'} \mid s \longrightarrow^* s' \text{ and } s' \vDash a\} \supseteq [\![q]\!]_s$.
- **RL-DECL-DECL.** Here $p \equiv p' \overset{a}{\searrow} p''$ and $q \equiv q' \overset{a}{\searrow} q''$. By the inductive hypothesis, we have $[\![p']\!]_s \supseteq [\![q']\!]_s$. Also, for any s' such that $s \longrightarrow^* s'$ and $s' \vDash a$, we have $a \vdash p'' \le q''$, so by the inductive hypothesis, $[\![p'']\!]_{s'} \supseteq [\![q'']\!]_{s'}$. So we have $[\![p' \overset{a}{\searrow} p'']\!]_s \supseteq [\![q' \overset{a}{\searrow} q'']\!]_s$.
- **RL-ERASE-E.** Here $p \equiv p' \overset{a}{\nearrow} p''$, and, by the inductive hypothesis, $[\![p']\!]_s \supseteq [\![q]\!]_s$ and $[\![p'']\!]_{s'} \supseteq [\![q]\!]_{s'}$ for any state s'. Suppose we have a pair $(s', \ell) \in [\![q]\!]_s$. Either (1) $(s', \ell) \in [\![p']\!]_s$ and $[s, s'] \nvDash a$, or (2) $(s', \ell) \in [\![p']\!]_s \cap [\![p'']\!]_{s''}$ for some s'' such that $s \longrightarrow^* s''$ and $[s, s'') \nvDash a$. If case (1) then $(s', \ell) \in [\![p]\!]_s$. If case (2) then by assumption $\vdash p'' \le q$, and by the inductive hypothesis, we have $[\![p'']\!]_{s''} \supseteq [\![q]\!]_{s''}$, and so $(s', \ell) \in [\![p]\!]_s$. Thus $[\![p]\!]_s \supseteq [\![q]\!]_s$ as required.
- **RL-ERASE-I.** Here $q \equiv p \overset{b}{\nearrow} q'$. Clearly, $[\![p]\!]_s \supseteq [\![p \overset{b}{\nearrow} q']\!]_s = [\![q]\!]_s$.
- **RL-ERASE-ERASE.** Here $p \equiv p' \overset{a}{\nearrow} p''$ and $q \equiv q' \overset{a}{\nearrow} q''$. By the inductive hypothesis, we have $[\![p']\!]_s \supseteq [\![q']\!]_s$ and for all s', $[\![p'']\!]_{s'} \supseteq [\![q'']\!]_{s'}$. Suppose we have a pair $(s', \ell) \in [\![q']\!]_s$. Either (1) $(s', \ell) \in [\![q']\!]_s$ and $[s, s'] \nvDash a$, or (2) $(s', \ell) \in [\![q]\!]_s \cap [\![q'']\!]_{s''}$ for some s'' such that $s \longrightarrow^* s''$ and $[s, s'') \nvDash a$. If case (1) then $(s', \ell) \in [\![p]\!]_s$. If case (2) then by assumption $\vdash p'' \le q''$, and by the inductive hypothesis, we have $[\![p'']\!]_{s''} \supseteq [\![q'']\!]_{s''}$, and so $(s', \ell) \in [\![p]\!]_s$. Thus $[\![p]\!]_s \supseteq [\![q]\!]_s$ as required.

∎

B Proof of Theorem 2

In this appendix we present the syntax and semantics of the language IMP^2_E, show that it is adequate to represent evaluation of two IMP_E programs, and show that type preservation in IMP^2_E implies Theorem 2.

This proof technique is based on Pottier and Simonet's [19] technique for showing noninterference in the ML programming language.

For notational convenience we write $\mathsf{reqErase}(p, \tau)$ as an abbreviation for $\mathsf{reqErase}(p, \tau[|\tau| - 1])$, where τ is a finite trace. For finite trace τ and trace τ' where $\tau[|\tau| - 1] \longrightarrow \tau'[0]$ we write $\tau\tau'$ for the trace obtained by appending τ' to τ.

B.1 Syntax and Semantics

The language IMP^2_E extends IMP_E with pair constructs for commands $(\!|\, c_1 \mid c_2 \,|\!)$, and integers $(\!|\, v_1 \mid v_2 \,|\!)$. The pair constructs represent different commands and integers that may arise in two different executions of a program. A command pair cannot be nested inside another command pair, but can otherwise appear nested at arbitrary depth. Integer pairs are used to track how memories differ in different executions of a program: memories in IMP^2_E are functions from variables to integers and integer pairs.

IMP^2_E syntax

$c ::=$		Commands
$\ldots$		IMP_E commands
$(\!|\, c_1 \mid c_2 \,|\!)$		Pair command

For an extended command c, let the projections $\lfloor c \rfloor_1$ and $\lfloor c \rfloor_2$ represent the two IMP_E commands that c encodes. The projection functions satisfy $\lfloor (\!|\, c_1 \mid c_2 \,|\!) \rfloor_i = c_i$, and are homomorphisms on other commands. Similarly for integer pairs, $\lfloor (\!|\, v_1 \mid v_2 \,|\!) \rfloor_i = v_i$. The projection functions are extended to memories, so that

$$\lfloor \sigma \rfloor_i(x) = \begin{cases} v & \text{if } \sigma(x) = v \\ v_i & \text{if } \sigma(x) = (\!|\, v_1 \mid v_2 \,|\!) \end{cases}$$

The evaluation of expressions are also extended, so that binary operations $\oplus$ are homomorphic on integer pairs. Thus, the evaluation of an expression e in a memory σ may be either an integer v or an integer pair $(\!|\, v_1 \mid v_2 \,|\!)$.

We extend configurations to triples $\langle c, \sigma \rangle_i$ for an index $i \in \{\bullet, 1, 2\}$. The index indicates if the command c and memory σ represent a pair of configurations ($\bullet$), or the left (1) or right (2) side of a pair of configurations. A configuration $\langle c, \sigma \rangle_i$ is well formed if $i \in \{1, 2\}$ implies that c does not contain any command pairs, and the image of σ does not contain any integer pairs.

The operational semantics of IMP^2_E are given in Figure 11, and extend the operational semantics of IMP_E. The rule OS-PAIR-LIFT allows the evaluation of either element of a command pair $(\!|\, c_1 \mid c_2 \,|\!)$. The rule OS-PAIR-SKIP removes a command pair when both elements of the pair have finished execution. The rule OS-PAIR-IF is used when the conditional of an **if** command evaluates to different values in the two executions, and as a result, a command pair is introduced, representing the different commands that each execution will evaluate. Note that this is the only way in which a command pair can be introduced into a configuration. For succinctness, this rule uses a ternary expression, $(v_i \neq 0)?c_0{:}c_1$, which is equal to c_0 if the predicate $v_i \neq 0$ is true, and to c_1 otherwise. The rule OS-PAIR-DECLASSIFY is used when the evaluation of conditions for a declassification differ in the two executions. The rule uses the evaluation of the product of the conditions, $e_0 \times \cdots \times e_k$, since this product will 0 if and only if there is some e_i that evaluates to zero.

The rules for IMP_E, given in Figure 5, are adapted by indexing each configuration with i to become rules for IMP^2_E. We assume that a premise of the form $\sigma(e) \neq 0$ means there is an integer v (not an integer pair) such that $\sigma(e) = v$ and $v \neq 0$. The utility functions $\mathit{update}(\cdot,\cdot,\cdot)$ and $\mathit{erasure}(\cdot)$ are adapted for IMP^2_E; the new versions, $\mathit{update}^2(\cdot,\cdot,\cdot)$ and $\mathit{erasure}^2(\cdot)$, are presented in Figure 5. For the adapted IMP_E rules, the version of the function to use depends upon the configuration index: the IMP^2_E versions if the index is $\bullet$; the IMP_E versions otherwise.

B.2 Adequacy

The language IMP^2_E is adequate for reasoning about the execution of two IMP_E programs. We show that execution of a IMP^2_E program is sound (a step taken by a IMP^2_E program corresponds to one or zero steps taken by its projections), and complete (given two IMP_E executions, there is a IMP^2_E execution whose projection agrees with at least one of them). We use $\rightarrow^=$ to denote the reflexive closure of the relation $\rightarrow$.

Lemma 1 (Soundness) *If $\langle c, \sigma \rangle_\bullet \rightarrow \langle c', \sigma' \rangle_\bullet$, then $\langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i \rangle \rightarrow^= \langle \lfloor c' \rfloor_i, \lfloor \sigma' \rfloor_i \rangle$ for $i \in \{1, 2\}$.*

Proof: By induction on the derivation $\langle c, \sigma \rangle_\bullet \rightarrow \langle c', \sigma' \rangle_\bullet$. The interesting cases are the new rules introduced for IMP^2_E: OS-PAIR-LIFT, OS-PAIR-SKIP, OS-PAIR-IF, and OS-PAIR-DECLASSIFY. For a reduction using OS-PAIR-LIFT, clearly one of the two projections takes a step, while the other projection remains unchanged. For OS-PAIR-SKIP, both projections remain unchanged. For both OS-PAIR-IF, and OS-PAIR-DECLASSIFY, both projections take a step. ∎

Lemma 2 (Stuck configurations) *If $\langle c, \sigma \rangle_\bullet$ is stuck (i.e., cannot be reduced and $c \neq$ **skip**), then $\langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i \rangle$ is stuck for some $i \in \{1, 2\}$.*

Proof: By structural induction on command c. ∎

Lemma 3 (Completeness) *If $\langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i \rangle \rightarrow^* \langle c'_i, \sigma'_i \rangle$ for $i \in \{1, 2\}$, then there exists a IMP^2_E configuration $\langle c', \sigma' \rangle_\bullet$ such that $\langle c, \sigma \rangle_\bullet \rightarrow^* \langle c', \sigma' \rangle_\bullet$ and $\langle \lfloor c' \rfloor_i, \lfloor \sigma' \rfloor_i \rangle = \langle c'_i, \sigma'_i \rangle$ for some $i \in \{1, 2\}$.*

Proof: Let $\tau_i = \langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i \rangle \ldots \langle c'_i, \sigma'_i \rangle$. Let n_i be the length of τ_i. For a IMP^2_E trace $\tau = \langle c, \sigma \rangle_\bullet \ldots \langle c', \sigma' \rangle_\bullet$, let $f_i(\tau)$ be n_i minus the number of reduction steps in τ that reduce the ith projection. Note that $f_i(\tau)$ is non-negative. Consider $g(\tau) = \min(f_1(\tau), f_2(\tau))$. If $g(\tau) = 0$, then τ is a trace that satisfies the conditions.

Suppose $g(\tau) > 0$. Consider the function

$$h(\tau) = (g(\tau),\ |f_1(\tau) - f_2(\tau)|,\ \mathit{numPairs}(\tau[|\tau| - 1]))$$

where $\mathit{numPairs}(\langle c, \sigma \rangle_\bullet)$ returns the number of pair commands in c. Note that all elements of the triple returned by $h(\tau)$ are non-negative. If we can extend τ by one step to a trace τ' such that $h(\tau') < h(\tau)$ under lexicographic ordering, then, by repeated applications, eventually we will produce a trace τ'' such that $g(\tau'') = 0$.

OS-PAIR-LIFT

$$\frac{\begin{array}{c}\langle c_i,\, \lfloor\sigma\rfloor_i\rangle_i \to \langle c_i',\, \sigma_i'\rangle_i \\ \{i,j\} = \{1,2\} \qquad c_j' = c_j \qquad \sigma_j' = \lfloor\sigma\rfloor_j \\ \sigma' = \lambda x.\ \begin{cases} 0 & \text{if } \mathsf{reqErase}(\Gamma(x), \lfloor\sigma\rfloor_1) \text{ and} \\ & \quad \mathsf{reqErase}(\Gamma(x), \lfloor\sigma\rfloor_2) \\ (\!|\, \sigma_1'(x) \mid \sigma_2'(x) \,|\!) & \text{if } \lfloor\sigma\rfloor_i(x) \neq \sigma_i'(x) \\ \sigma(x) & \text{otherwise} \end{cases}\end{array}}{\langle (\!|\, c_1 \mid c_2 \,|\!),\, \sigma\rangle_\bullet \to \langle (\!|\, c_1' \mid c_2' \,|\!),\, \sigma'\rangle_\bullet}$$

OS-PAIR-SKIP

$$\frac{}{\langle (\!|\, \mathbf{skip} \mid \mathbf{skip} \,|\!),\, \sigma\rangle_\bullet \to \langle \mathbf{skip},\, \sigma\rangle_\bullet}$$

OS-PAIR-IF

$$\frac{\begin{array}{c}\sigma(e) = (\!|\, v_1 \mid v_2 \,|\!) \\ c_i' = (v_i \neq 0)\ ?\ c_0 : c_1\end{array}}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1,\, \sigma\rangle_\bullet \to \langle (\!|\, c_1' \mid c_2' \,|\!),\, \sigma\rangle_\bullet}$$

OS-PAIR-DECLASSIFY

$$\frac{\begin{array}{c}\sigma(e_0 \times \cdots \times e_k) = (\!|\, v_1 \mid v_2 \,|\!) \\ v_i' = (v_i \neq 0)\ ?\ \lfloor\sigma(e)\rfloor_i : 0 \\ \sigma' = \mathit{update}^2(\sigma, x, (\!|\, v_1' \mid v_2' \,|\!))\end{array}}{\langle x := \mathbf{declassify}(e, L_f\ \mathbf{to}\ L_t\ \mathbf{using}\ e_0, \ldots, e_k),\, \sigma\rangle_\bullet \to \langle \mathbf{skip},\, \sigma'\rangle_\bullet}$$

$$\mathit{update}^2(\sigma, y, w) = \lambda x.\ \begin{cases} 0 & \text{if } \forall i \in \{1,2\}.\ \mathsf{reqErase}(\Gamma(x), \sigma_i') \\ (\!|\, \sigma_1'(x) \mid \sigma_2'(x) \,|\!) & \text{if } \exists i \in \{1,2\}.\ \lfloor\sigma\rfloor_i(x) \neq \sigma_i'(x) \\ & \quad \text{and } \exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(x), \sigma_i') \\ \sigma(x) & \text{otherwise} \end{cases}$$

$$\text{where } \sigma_i' = \mathit{update}(\lfloor\sigma\rfloor_i, y, \lfloor w\rfloor_i)$$

$$\mathit{erasure}^2(\sigma) = \lambda x.\ \begin{cases} 0 & \text{if } \forall i \in \{1,2\}.\ \mathsf{reqErase}(\Gamma(x), \sigma_i') \\ (\!|\, \sigma_1'(x) \mid \sigma_2'(x) \,|\!) & \text{if } \exists i \in \{1,2\}.\ \lfloor\sigma\rfloor_i(x) \neq \sigma_i'(x) \\ & \quad \text{and } \exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(x), \sigma_i') \\ \sigma(x) & \text{otherwise} \end{cases}$$

$$\text{where } \sigma_i' = \mathit{erasure}(\lfloor\sigma\rfloor_i)$$

Figure 11: Operational semantics of IMP_E^2

We now show how to extend τ to a trace τ' such that $h(\tau') < h(\tau)$. By assumption, $g(\tau) > 0$, so neither τ_1 or τ_2 is stuck. By Lemma 2, we can extend τ by one more step, producing trace τ'. By Lemma 1, either $f_i(\tau') = f_i(\tau) - 1$ for some $i \in \{1,2\}$, or $f_i(\tau') = f_i(\tau)$ for all $i \in \{1,2\}$. If the former, then $h(\tau') < h(\tau)$. If the latter, then the rule OS-PAIR-SKIP was used in the reduction, and the last configuration of τ' has one fewer pair command than the last configuration of τ, and so $h(\tau') < h(\tau)$. ∎

B.3 Type preservation

We extend the type system to type IMP_E^2 commands and configurations. The typing judgment for commands is now of the form $\tau, pc, \Gamma \vdash c\ \mathsf{com}$, where τ is an execution trace. If $\tau, pc, \Gamma \vdash c\ \mathsf{com}$, then command c is well-typed with typing context Γ and program counter policy pc at the program point when trace τ has been produced. Typing rules for IMP_E (given in Figure 7) are made typing rules for IMP_E^2 by adding the additional typing parameter τ to each rule. Similarly, the judgment $\tau, pc, \Gamma \vdash \langle c,\, \sigma\rangle_\bullet\ \mathsf{config}$ means that configuration $\langle c,\, \sigma\rangle_\bullet$ is well-typed with typing context Γ and program counter policy pc at the program point when trace τ has been produced.

$$\frac{}{p \leq_{\langle c,\, \sigma\rangle} p} \qquad \frac{\begin{array}{cc}\tau = \tau'\langle c,\, \sigma\rangle & \neg\mathsf{reqErase}(p', \tau') \\ p \leq_{\tau'} p' & [\![p']\!]_\sigma \supseteq [\![q]\!]_\sigma\end{array}}{p \leq_\tau q}$$

Figure 13: Inference rules for $p \leq_\tau q$

The two new typing rules, shown in Figure 12, make use of the predicate $\mathit{protected}(p, \tau)$. Informally, if for policy p the predicate $\mathit{protected}(p, \tau)$ is true, then the program input may have flowed through the program, and now be labeled with the policy p. Thus, this predicate depends on the execution trace τ. Note that the premises for the typing rule for pairs, T-PAIR, uses the typing judgments for IMP_E, i.e., without the trace τ. This is because well-formed commands do not have nested command pairs.

To formalize how program input may be relabeled with different policies, we use the *extended relabeling* relation

$$\textsc{T-Pair}\quad \frac{\vdash pc \le pc' \qquad \mathit{protected}(pc', \tau) \qquad \neg\mathsf{reqErase}(pc', \lfloor\tau\rfloor_1) \qquad pc', \Gamma \vdash c_1\ \mathsf{com} \qquad \neg\mathsf{reqErase}(pc', \lfloor\tau\rfloor_2) \qquad pc', \Gamma \vdash c_2\ \mathsf{com}}{\tau, pc, \Gamma \vdash (\!| c_1 \mid c_2 |\!)\ \mathsf{com}}$$

$$\textsc{T-Config}\quad \frac{\tau, pc, \Gamma \vdash c\ \mathsf{com} \qquad \forall x \in \mathit{Vars}.\ (\sigma(x) = (\!| v_1 \mid v_2 |\!)) \Rightarrow \mathit{protected}(\Gamma(x), \tau)}{\tau, pc, \Gamma \vdash \langle c,\ \sigma\rangle_\bullet\ \mathsf{config}}$$

Figure 12: Typing rules for IMP_E^2

$p \le_\tau q$. For policies p and q and finite trace τ, if $p \le_\tau q$, then input in $\tau[0]$, the initial configuration of trace τ, labeled with policy p can influence information labeled with policy q in final configuration of τ. Inference rules for this relation are given in Figure 13.

More formally, we define $\mathit{protected}(p, \tau)$ as

$$\begin{aligned}\mathit{protected}(p,\tau) \triangleq (&\neg\mathsf{reqErase}(p, \lfloor\tau\rfloor_1) \vee \\ &\neg\mathsf{reqErase}(p, \lfloor\tau\rfloor_2)) \ \wedge \\ &\forall i \in \{1,2\}.\ \neg\mathsf{reqErase}(p, \lfloor\tau\rfloor_i) \Rightarrow \\ &\Gamma(x) \le_{\lfloor\tau\rfloor_i} p\end{aligned}$$

where the variable x is the variable in which program input is placed.

The extended relabeling relation has a nice property with respect to the semantics of policies. If $\tau = \langle c,\ \sigma\rangle \ldots \langle c',\ \sigma'\rangle$ and $p \le_\tau q$ then the semantics of q in $\langle c',\ \sigma'\rangle$ are a subset of the semantics of p in $\langle c,\ \sigma\rangle$. We prove this using the following lemma.

Lemma 4 *If* $\langle c,\ \sigma\rangle \to \langle c',\ \sigma'\rangle$ *and* $\neg\mathsf{reqErase}(p, \sigma)$ *then* $[\![p]\!]_{\langle c',\ \sigma'\rangle} \subseteq [\![p]\!]_{\langle c,\ \sigma\rangle}$.

Proof: By induction on the structure of p. ∎

Property 2 *If* $\tau = \langle c,\ \sigma\rangle \ldots \langle c',\ \sigma'\rangle$ *and* $p \le_\tau q$ *then* $[\![q]\!]_{\langle c',\ \sigma'\rangle} \subseteq [\![p]\!]_{\langle c,\ \sigma\rangle}$.

Proof: By induction on the derivation of $p \le_\tau q$, using Lemma 4. ∎

A well-typed IMP_E^2 program tracks information flow from the initial input. The execution of a IMP_E^2 program preserves typing. This key theorem will allow us to prove that well-typed IMP_E^2 satisfied noninterference according to policy.

Theorem 4 (Type preservation) *Let* Γ *be a well-formed typing context, and* c_0 *a* IMP_E *command,* c, c' IMP_E^2 *commands, and* $\sigma_0, \sigma, \sigma'$ IMP_E^2 *memories such that* $\sigma_0 = \mathit{erasure}^2(\sigma_0)$ *and* $\langle c_0,\ \sigma_0\rangle_\bullet \to^* \langle c,\ \sigma\rangle_\bullet \to \langle c',\ \sigma'\rangle_\bullet$.

Let $\tau = \langle c_0,\ \sigma_0\rangle_\bullet \ldots \langle c,\ \sigma\rangle_\bullet$*, and let* $\tau' = \tau\langle c',\ \sigma'\rangle_\bullet$.

If $\tau, \bot, \Gamma \vdash \langle c,\ \sigma\rangle_\bullet \mathsf{config}$ *then* $\tau', \bot, \Gamma \vdash \langle c',\ \sigma'\rangle_\bullet \mathsf{config}$.

Before we prove Theorem 4, we first state and prove a series of useful lemmas.

The first two lemmas relate to the program counter policy. If a program is well-typed for some program counter policy pc, then it is also well-typed for any weaker program counter policy, and also, any variable x that is updated in the next step satisfies $\vdash pc \le \Gamma(x)$.

Lemma 5 *If* $\vdash pc \le pc'$ *and* $\tau, pc', \Gamma \vdash c\ \mathsf{com}$ *then* $\tau, pc, \Gamma \vdash c\ \mathsf{com}$.

Proof: By induction on $\tau, pc', \Gamma \vdash c\ \mathsf{com}$. ∎

Lemma 6 *Let* Γ *be a well-formed typing context,* τ *a trace,* $i \in \{1, 2, \bullet\}$*,* c, c' *commands, and* $\sigma, sigma'$ *memories such that* $\langle c,\ \sigma\rangle_i \to \langle c',\ \sigma'\rangle_i$ *and* $\tau, pc, \Gamma \vdash c\ \mathsf{com}$*. For all* $x \in \mathit{Vars}$*, if* $\sigma(x) \ne \sigma'(x)$ *then* $\vdash pc \le \Gamma(x)$.

Moreover, if the execution step assigned some variable y*, then for all* $x \in \mathit{Vars}$*, if* $\sigma(x) \ne \sigma'(x)$ *then* $\vdash \Gamma(y) \le \Gamma(x)$.

Proof: By induction on the derivation of $\langle c,\ \sigma\rangle_i \to \langle c',\ \sigma'\rangle_i$. The only way the memory can change is by assigning some variable y the value (or pair value) v, via the utility function $\mathit{update}(\sigma, y, v)$.

If $x = y$, then the appropriate typing rule (OS-ASSIGN, OS-DECLASSIFY, or OS-PAIR-DECLASSIFY) requires that $\vdash pc \le \Gamma(y)$ and $\Gamma(y) = \Gamma(x)$. If $x \ne y$, then, considering the definition of $\mathit{erasure}(\sigma[y \mapsto v])$ there must be some k such that $\sigma'(x) = \sigma_k(x) \ne \sigma(x)$. By induction on k, we can show that for any variable z, if $\sigma_k(z) \ne \sigma(z)$, then $\vdash pc \le \Gamma(y)$ and $\vdash \Gamma(y) \le \Gamma(z)$. The base case $k = 0$ follows from the typing rules requiring $\vdash pc \le \Gamma(y)$. The inductive case is that if $\neg\mathsf{reqErase}(z, \sigma)$ but $\mathsf{reqErase}(z, \sigma_{k+1})$, then there must be some variable z' such that $\sigma(z') \ne \sigma_k(z') = \sigma'(z')$, z' appears in an expression in $\mathsf{eraseConds}(\Gamma(z))$. By the induction hypothesis, we have $\vdash pc \le \Gamma(y)$ and $\vdash \Gamma(y) \le \Gamma(z')$. Since Γ is well-formed, we have $\vdash \Gamma(z') \le \Gamma(z)$, and so $\vdash pc \le \Gamma(y)$ and $\vdash \Gamma(y) \le \Gamma(z)$ as required. ∎

To prove type preservation in IMP_E^2, it is helpful to know that IMP_E also preserves types.

Lemma 7 (Type preservation for IMP_E) *Let* Γ *be a well-formed typing context, and* c *a* IMP_E *command, and* σ *a* IMP_E *memory, and* pc *a policy such that* $pc, \Gamma \vdash c\ \mathsf{com}$*. For* $i \in \{1, 2\}$*, if* $\langle c,\ \sigma\rangle_i \to \langle c',\ \sigma'\rangle_i$ *then* $pc, \Gamma \vdash c'\ \mathsf{com}$.

Proof: By induction on $\langle c,\ \sigma\rangle_i \to \langle c',\ \sigma'\rangle_i$, using Lemma 5 applied to IMP_E type judgments. ∎

The following series of lemmas are related to showing that nice properties hold for the utility functions $\mathit{update}^2(\cdot,\cdot,\cdot)$ and $\mathit{erasure}^2(\cdot)$. Several of them are concerned with memories σ that satisfy $\sigma = \mathit{erasure}^2(\sigma)$. We call such memories *consistent*, as they are consistent with erasure requirements: $\forall i \in \{1, 2\}.\ \forall x \in Vars.\ \mathsf{reqErase}(\Gamma(x), \lfloor\sigma\rfloor_i) \Rightarrow \lfloor\sigma(x)\rfloor_i = 0$.

The IMP^2_E utility functions $\mathit{update}^2(\cdot,\cdot,\cdot)$ and $\mathit{erasure}^2(\cdot)$ agree with their IMP_E versions.

Lemma 8 *Let σ and σ' be IMP^2_E memories. If $\sigma' = \mathit{update}^2(\sigma, x, (\!|\, v_1 \mid v_2 \,|\!))$ for some variable x and values v_1 and v_2, then for all $i \in \{1,2\}$, $\lfloor \sigma' \rfloor_i = \mathit{update}(\lfloor \sigma \rfloor_i, x, v_i)$. Similarly, if $\sigma' = \mathit{erasure}^2(\sigma)$, then for all $i \in \{1,2\}$, $\lfloor \sigma' \rfloor_i = \mathit{erasure}(\lfloor \sigma \rfloor_i)$*

Proof: Suppose $\sigma' = \mathit{update}^2(\sigma, x, (\!|\, v_1 \mid v_2 \,|\!))$ for some variable x and values v_1 and v_2. Let $\sigma'_i = \mathit{update}(\lfloor \sigma \rfloor_i, x, v_i)$. Let y be a variable. If $\mathsf{reqErase}(\Gamma(y), \sigma'_1)$ and $\mathsf{reqErase}(\Gamma(y), \sigma'_2)$ then $\sigma'_i(y) = 0 = \lfloor \sigma' \rfloor_i(y)$ as required. If $\sigma'_1(y) = \lfloor \sigma \rfloor_1(y)$ and $\sigma'_2(y) = \lfloor \sigma \rfloor_2(y)$ then $\lfloor \sigma'(y) \rfloor_i = \lfloor \sigma(y) \rfloor_i = \sigma'_i(y)$ as required. Otherwise, $\sigma'(y) = (\!|\, \sigma'_1(y) \mid \sigma'_2(y) \,|\!)$, and so $\lfloor \sigma(y) \rfloor_i = \sigma'_i(y)$ as required.

Now suppose $\sigma' = \mathit{erasure}^2(\sigma)$. Let $\sigma'_i = \mathit{erasure}(\lfloor \sigma \rfloor_i)$. Let y be a variable. If $\mathsf{reqErase}(\Gamma(y), \sigma'_1)$ and $\mathsf{reqErase}(\Gamma(y), \sigma'_2)$ then $\sigma'_i(y) = 0 = \lfloor \sigma' \rfloor_i(y)$ as required. If $\sigma'_1(y) = \lfloor \sigma \rfloor_1$ and $\sigma'_2(y) = \lfloor \sigma \rfloor_2$ then $\lfloor \sigma'(y) \rfloor_i = \lfloor \sigma(y) \rfloor_i = \sigma'_i(y)$ as required. Otherwise, $\sigma'(y) = (\!|\, \sigma'_1(y) \mid \sigma'_2(y) \,|\!)$, and so $\lfloor \sigma(y) \rfloor_i = \sigma'_i(y)$ as required. ∎

The utility function $\mathit{update}^2(\cdot,\cdot,\cdot)$ establishes a consistent memory.

Lemma 9 *Let σ and σ' be IMP^2_E memories such that $\sigma' = \mathit{update}^2(\sigma, x, w)$ for some variable x and value w. Then $\sigma' = \mathit{erasure}^2(\sigma')$*

Proof: Note that for IMP_E memories σ_0 and σ'_0 if $\sigma'_0 = \mathit{update}(\sigma_0, x, v)$ for some variable x and value v, then $\sigma'_0 = \mathit{erasure}(\sigma'_0)$. This follows easily from the definition of $\mathit{update}(\sigma_0, x, v)$ and the idempotency of $\mathit{erasure}(\cdot)$.

Let $\sigma'_i = \mathit{erasure}(\lfloor \sigma' \rfloor_i)$. We have

$$\begin{aligned}\sigma'_i &= \mathit{erasure}(\lfloor \sigma' \rfloor_i)\\ &= \mathit{erasure}(\lfloor \mathit{update}^2(\sigma, x, w) \rfloor_i)\\ &= \mathit{erasure}(\mathit{update}(\lfloor \sigma \rfloor_i, x, \lfloor w \rfloor_i))\\ &= \mathit{update}(\lfloor \sigma \rfloor_i, x, \lfloor w \rfloor_i).\end{aligned}$$

Let y be a variable. Consider $\mathit{erasure}^2(\sigma')(y)$. If $\forall i \in \{1,2\}.\ \mathsf{reqErase}(\Gamma(y), \sigma'_i)$ then from the definition of $\mathit{update}^2(\sigma, x, w)$ we have $\sigma'(y) = 0 = \mathit{erasure}^2(\sigma')(y)$. Similarly, if $\exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(y), \sigma'_i)$ and $\exists i \in \{1,2\}.\ \lfloor \sigma \rfloor_i(y) \neq \sigma'_i(y)$, then from the definition of $\mathit{update}^2(\sigma, x, w)$ we have $\sigma'(y) = (\!|\, \sigma'_1(y) \mid \sigma'_2(y) \,|\!) = \mathit{erasure}^2(\sigma')(y)$. Finally, if $\exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(y), \sigma'_i)$ and $\forall i \in \{1,2\}.\ \lfloor \sigma \rfloor_i(y) = \sigma'_i(y)$, then from the definition of $\mathit{update}^2(\sigma, x, w)$ we have $\sigma'(y) = \sigma(y) = \mathit{erasure}^2(\sigma')(y)$. ∎

The semantics of IMP^2_E preserves consistent memories.

Lemma 10 *Let Γ be a well-formed typing context, and c_0 a IMP_E command, c a IMP^2_E command, and σ_0, σ IMP^2_E memories such that $\sigma_0 = \mathit{erasure}^2(\sigma_0)$ and $\tau = \langle c_0, \sigma_0 \rangle_\bullet \longrightarrow^* \langle c, \sigma \rangle_\bullet$. Then $\sigma = \mathit{erasure}^2(\sigma)$.*

Proof: By induction on $\longrightarrow$. When a step does not change the memory, this is trivial. For OS-ASSIGN, OS-DECLASSIFY, and OS-PAIR-DECLASSIFY, the result follows from the idempotency of $\mathit{erasure}^2(\cdot)$. For OS-PAIR-LIFT, it follows from the idempotency of $\mathit{erasure}(\cdot)$. ∎

For consistent memories σ, a variable x will map to a pair value in a σ only if the policy $\Gamma(x)$ does not require erasure in at least one of the projections. Equivalently, if $\Gamma(x)$ requires erasure in both projections, then $\mathit{sigma}(x)$ will not be a pair value.

Lemma 11 *For all IMP^2_E memories σ, and all variables $x \in \mathit{Vars}$, if $\sigma = \mathit{erasure}^2(\sigma)$ and $\sigma(x) = (\!|\, v_1 \mid v_2 \,|\!)$, then $\neg\mathsf{reqErase}(\Gamma(x), \lfloor \sigma \rfloor_i)$ for some $i \in \{1,2\}$.*

Proof: Immediate from the definition of $\mathit{erasure}^2(\sigma)$. ∎

For consistent memories σ, if there is a variable x that maps to a pair value, then there is some variable y such that y also maps to a pair value, $\Gamma(y)$ does not require erasure in either projection, and information is allowed to flow from y to x. The proof of this lemma uses the well-foundedness of the overwrite dependency relation $\prec_\Gamma$.

Lemma 12 *For any IMP^2_E memory σ, and variable x, if $\sigma = \mathit{erasure}^2(\sigma)$ and $\sigma(x) = (\!|\, v_1 \mid v_2 \,|\!)$, then there is a variable y such that $\sigma(y) = (\!|\, v'_1 \mid v'_2 \,|\!)$ and $\neg\mathsf{reqErase}(\Gamma(y), \lfloor \tau \rfloor_i)$ for all $i \in \{1,2\}$ and $\vdash \Gamma(y) \le \Gamma(x)$*

Proof: If $\neg\mathsf{reqErase}(\Gamma(x), \lfloor \sigma \rfloor_i)$ for all $i \in \{1,2\}$, then we are done. If not, then by Lemma 11, $\neg\mathsf{reqErase}(\Gamma(x), \lfloor \sigma \rfloor_i)$ for some $i \in \{1,2\}$. This means there is some expression $e \in \mathsf{eraseConds}(\Gamma(x))$ such that $\sigma(e)$ is a pair value, and so there is some variable x_0 that appears in e such that $\sigma(x_0)$ is a pair value. Since Γ is well-formed, we have $\vdash \Gamma(x_0) \le \Gamma(x)$. Note that $x_0 \prec_\Gamma x$. If $\neg\mathsf{reqErase}(\Gamma(x_0), \lfloor \tau \rfloor_i)$ for all $i \in \{1,2\}$, then we are done. Otherwise, we repeat the argument, forming a chain $x_0, x_1, \ldots$ such that $x_{k+1} \prec_\Gamma x_k$ and $\vdash \Gamma(x_{k+1}) \le \Gamma(x_k)$. Since Γ is well-formed, the relation $\prec_\Gamma$ is well-founded, and thus eventually a variable x_n will be found such that $\sigma(x_n)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(x_n), \lfloor \tau \rfloor_i)$ for all $i \in \{1,2\}$. ∎

The next two lemmas are concerned with the preservation of predicates $\mathit{protected}(p, \tau)$ and $\neg\mathsf{reqErase}(p, \lfloor \tau \rfloor_i)$ when the trace τ is extended. The first claims that if τ is extended by one step to τ' but the memory is not changed in that step, then $\mathit{protected}(p, \tau)$ implies $\mathit{protected}(p, \tau')$. The second lemma claims that if a IMP^2_E command c is well-typed for a trace τ, and trace τ' satisfies all $\mathit{protected}(\cdot,\cdot)$ and $\neg\mathsf{reqErase}(\cdot,\cdot)$ predicates that τ does, then c is well-typed for τ'.

Lemma 13 *Let Γ be a well-formed typing context, and c_0, c IMP_E commands, and σ_0, σ IMP^2_E memories such that $\sigma_0 = \mathit{erasure}^2(\sigma_0)$ and $\langle c_0, \sigma_0 \rangle_\bullet \longrightarrow^* \langle c, \sigma \rangle_\bullet$. Suppose $\langle c, \sigma \rangle_\bullet \longrightarrow \langle c', \sigma \rangle_\bullet$. Let $\tau = \langle c_0, \sigma_0 \rangle_\bullet \ldots \langle c, \sigma \rangle_\bullet$, and let $\tau' = \tau \langle c', \sigma \rangle_\bullet$. Then for all policies p, if $\mathit{protected}(p, \tau)$ then $\mathit{protected}(p, \tau')$.*

Proof: Suppose $\mathit{protected}(p, \tau)$. We need to show that either $\neg\mathsf{reqErase}(p, \lfloor \tau' \rfloor_1)$ or $\neg\mathsf{reqErase}(p, \lfloor \tau' \rfloor_2)$ and that for $i \in \{1, 2\}$ if $\neg\mathsf{reqErase}(p, \lfloor \sigma \rfloor_i)$ then $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} p$.

First note that the final memories of τ and τ' are identical, and so $\neg\mathsf{reqErase}(p, \lfloor \tau \rfloor_i)$ if and only if $\neg\mathsf{reqErase}(p, \lfloor \tau' \rfloor_i)$.

Since $\mathit{protected}(p, \tau)$, either $\neg\mathsf{reqErase}(p, \lfloor \tau \rfloor_1)$ or $\neg\mathsf{reqErase}(p, \lfloor \tau \rfloor_2)$, and so either $\neg\mathsf{reqErase}(p, \lfloor \tau' \rfloor_1)$ or $\neg\mathsf{reqErase}(p, \lfloor \tau' \rfloor_2)$.

Suppose for some i we have $\neg\mathsf{reqErase}(p, \lfloor \tau' \rfloor_i)$. Then $\neg\mathsf{reqErase}(p, \lfloor \tau \rfloor_i)$, and since $\mathit{protected}(p, \tau)$, $\Gamma(x) \leq_{\lfloor \tau \rfloor_i} p$. By the inference rules for extended relabeling, we can conclude $\mathit{protected}(p, \tau')$. ∎

Lemma 14 *Let τ and τ' be IMP^2_E traces, Γ a well-formed context, pc a policy, and c a IMP^2_E command such that $\tau, pc, \Gamma \vdash c\ \mathsf{com}$. If for all policies p we have $\mathit{protected}(p, \tau) \Rightarrow \mathit{protected}(p, \tau')$ and $\neg\mathsf{reqErase}(p, \lfloor \tau \rfloor_i) \Rightarrow \neg\mathsf{reqErase}(p, \lfloor \tau' \rfloor_i)$, then $\tau', pc, \Gamma \vdash c\ \mathsf{com}$.*

Proof: By induction on the derivation of $\tau, pc, \Gamma \vdash c\ \mathsf{com}$, the only interesting case being T-PAIR. ∎

Pair commands are introduced into a configuration only when an **if** command is executed, and the conditional expression evaluates to a pair value. This restricts where pair commands may appear.

Lemma 15 *Let Γ be a well-formed context, c a IMP_E command, and σ a IMP^2_E memory. For any configuration $\langle c', \sigma' \rangle_\bullet$ such that $\langle c, \sigma \rangle_\bullet \rightarrow^* \langle c', \sigma' \rangle_\bullet$, and any sequence $d_0; d_1$ that is a sub-command of c', the command d_1 does not contain any pair commands.*

Proof: By induction on $\langle c, \sigma \rangle_\bullet \rightarrow^* \langle c', \sigma' \rangle_\bullet$. ∎

Using these lemmas, we can now prove that IMP^2_E preserves typing.

Proof of Theorem 4: Proof is by induction on the judgment $\langle d, \sigma \rangle_\bullet \rightarrow \langle d', \sigma' \rangle_\bullet$. Let $\tau = \langle c_0, \sigma_0 \rangle_\bullet \ldots \langle c, \sigma \rangle_\bullet$, and let $\tau' = \tau \langle c', \sigma' \rangle_\bullet$. Note that by Lemma 10, we have $\sigma = \mathit{erasure}^2(\sigma)$ and $\sigma' = \mathit{erasure}^2(\sigma')$.

- OS-SKIP. Here $d = \mathbf{skip}; d'$. Since the memory is unchanged, by Lemma 13 and Lemma 14 we have $\tau', \bot, \Gamma \vdash d'\ \mathsf{com}$.

- OS-ASSIGN. Here $d = y := e$ and $d' = \mathbf{skip}$. By the typing rule for **skip**, we have $\tau', \bot, \Gamma \vdash d'\ \mathsf{com}$. We need to show that $\forall z \in \mathit{Vars}.\ (\sigma'(z) = (\!| v_1 \mid v_2 |\!)) \Rightarrow \mathit{protected}(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z) = (\!| v_1 \mid v_2 |\!)$. By Lemma 11, either $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_1)$ or $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_i)$ then $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(z)$, where x is the input variable. Suppose $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_i)$.

  First, suppose $\sigma(z) = \sigma'(z)$. If $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau \rfloor_i)$, then, by the inference rules for the extended relabeling relation, we can conclude that $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(z)$, and we are done. Otherwise $\mathsf{reqErase}(\Gamma(z), \lfloor \tau \rfloor_i)$, and so by Lemma 11 and Lemma 12, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor \tau \rfloor_i)$ and $\vdash \Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c, \sigma \rangle_\bullet\ \mathsf{config}$, we have $\mathit{protected}(\Gamma(w), \tau)$, and so $\Gamma(x) \leq_{\lfloor \tau \rfloor_i} \Gamma(w)$, and thus, by the inference rules for the extended relabeling relation, $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(z)$ as required.

  Otherwise, $\sigma(z) \neq \sigma'(z)$ and so either $z = y$, or z was updated by the utility function $\mathit{erasure}(\cdot)$.

  If $z = y$, then $\sigma(e)$ is a pair value, since $\sigma'(z)$ is a pair value. Thus there must be a variable w that appears in e such that $\sigma(w)$ is a pair, and $\vdash \Gamma(w) \leq \Gamma(z)$ (by the typing rule for assignment). By Lemma 12, there is a variable w' such that $\sigma(w')$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w'), \lfloor \tau \rfloor_i)$ and $\vdash \Gamma(w') \leq \Gamma(w)$ and $\vdash \Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c, \sigma \rangle_\bullet\ \mathsf{config}$, we have $\mathit{protected}(\Gamma(w'), \tau)$, and so $\Gamma(x) \leq_{\lfloor \tau \rfloor_i} \Gamma(w')$, and thus, by the inference rules for the extended relabeling relation, $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(z)$ as required.

  Finally, if $\sigma(z) \neq \sigma'(z)$ and $z \neq y$, then z was updated by the utility function $\mathit{erasure}(\cdot)$. Note that this means $\Gamma(z)$ requires erasure on at least one of the two projections. By Lemma 12, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor \tau \rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor \tau \rfloor_2)$ and $\vdash \Gamma(w) \leq \Gamma(z)$. Since $\Gamma(w)$ does not require erasure in either projection, w was not updated by the utility function $\mathit{erasure}(\cdot)$. Thus, by the previous cases, we have $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(z)$ as required.

- OS-SEQUENCE. Here $d = d_1; d_2$ and $d' = d_1'; d_2$. By the inductive hypothesis, we have $\tau', \bot, \Gamma \vdash d_1'\ \mathsf{com}$, and that $\forall x \in \mathit{Vars}.\ (\sigma'(x) = (\!| v_1 \mid v_2 |\!)) \Rightarrow \mathit{protected}(\Gamma(x), \tau')$. We need to show that $\tau', \bot, \Gamma \vdash d_2\ \mathsf{com}$. We do this by an easy induction on the derivation of $\tau, \bot, \Gamma \vdash d_2\ \mathsf{com}$ which relies on the fact that, by Lemma 15, the command d_2 cannot contain a command pair $(\!| d_3 \mid d_4 |\!)$. Thus we have $\tau', \bot, \Gamma \vdash \langle d_1'; d_2, \sigma' \rangle_\bullet\ \mathsf{config}$ as required.

- OS-IF. Here $d = \mathbf{if}\ e\ \mathbf{then}\ d_0\ \mathbf{else}\ d_1$ and $d' = d_i$ for some $i \in \{0, 1\}$. By the typing rule for **if**, and Lemma 5, we have $\tau, \bot, \Gamma \vdash d'\ \mathsf{com}$. Since the memory is unchanged, by Lemma 13 and Lemma 14 we have $\tau', \bot, \Gamma \vdash d'\ \mathsf{com}$.

- OS-WHILE. Here $d = \mathbf{while}\ e\ \mathbf{do}\ d_1$ and $d' = \mathbf{if} e \mathbf{then} d_1; \mathbf{while} e \mathbf{do} d_1 \mathbf{else} \mathbf{skip}$. Since $\tau, \bot, \Gamma \vdash d \mathsf{com}$, we have $\Gamma \vdash e : p_e\ \mathsf{exp}$, and there exists some policy pc' such that $\vdash \bot \leq pc'$, and $\vdash p_e \leq pc'$, and $\tau, pc', \Gamma \vdash d_1\ \mathsf{com}$. From this we can derive $\tau, pc', \Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ d_1\ \mathsf{com}$, and thus $\tau, pc', \Gamma \vdash d'\ \mathsf{com}$, and so by Lemma 5, we have $\tau, \bot, \Gamma \vdash d'\ \mathsf{com}$. Since the memory is unchanged, by Lemma 13 and Lemma 14 we have $\tau', \bot, \Gamma \vdash d'\ \mathsf{com}$.

- OS-DECLASSIFY. Here $d = y :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$ and $d' =$ **skip**. By the typing rule for **skip**, we have $\tau', \bot, \Gamma \vdash d'$ com. We need to show that $\forall z \in Vars.\ (\sigma'(z) = (\!| v_1 \mid v_2 |\!)) \Rightarrow protected(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z) = (\!| v_1 \mid v_2 |\!)$. By Lemma 11, either $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_1)$ or $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$ then $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$. Suppose $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$.

  First, suppose $\sigma(z) = \sigma'(z)$. The reasoning is exactly the same as the analogous subcase in OS-ASSIGN.

  Otherwise, $\sigma(z) \neq \sigma'(z)$ and so either $z = y$, or z was updated by the utility function $erasure(\cdot)$.

  If $z = y$, then $\sigma(e)$ is a pair value, since $\sigma'(z)$ is a pair value. (This implies that all conditions e_i were satisfied, as otherwise, z is updated with the non-pair value 0.) The reasoning in this case is exactly the same as the analogous subcase in OS-ASSIGN.

  Finally, if $\sigma(z) \neq \sigma'(z)$ and $z \neq y$, then z was updated by the utility function $erasure(\cdot)$. The reasoning here is exactly the same as the analogous subcase in OS-ASSIGN.

- OS-PAIR-SKIP. Immediate by typing rule for **skip**.

- OS-PAIR-LIFT. Here $d = (\!| d_1 \mid d_2 |\!)$ and $d' = (\!| d_1' \mid d_2' |\!)$. If $\sigma' = \sigma$, then by Lemma 13, we have $\tau', \bot, \Gamma \vdash d'$ com. So suppose $\sigma' \neq \sigma$.

  Since $\tau, \bot, \Gamma \vdash d$ com, there is a pc' such that $protected(pc', \tau)$, $\neg\mathsf{reqErase}(pc', \tau)$, and $pc', \Gamma \vdash d_i$ com, for $i \in \{1, 2\}$. By Lemma 7, we have $pc', \Gamma \vdash d_i'$ com, for $i \in \{1, 2\}$.

  Without loss of generality, assume that it is the left execution that makes progress, and thus $d_2 = d_2'$. Note that since the left projection made progress, the right projection was unchanged, and so $\lfloor\sigma'\rfloor_2 = \lfloor\sigma\rfloor_2$, and thus $\neg\mathsf{reqErase}(pc', \lfloor\sigma'\rfloor_2)$.

  If $\neg\mathsf{reqErase}(pc', \lfloor\sigma'\rfloor_1)$, then we can easily show that $protected(pc', \tau')$, and so we have $\tau', \bot, \Gamma \vdash d'$ com, as required.

  Otherwise, suppose $\mathsf{reqErase}(pc', \lfloor\sigma'\rfloor_1)$. Since $\sigma' \neq \sigma$, there must have been either an assignment or a declassification, updating some variable y. By the definition of $update(\cdot, \cdot, \cdot)$, this means that $\neg\mathsf{reqErase}(\Gamma(y), \lfloor\tau\rfloor_1)$, and by the well-formedness of Γ, $\neg\mathsf{reqErase}(\Gamma(y), \lfloor\tau'\rfloor_1)$ (since whether $\Gamma(y)$ requires erasure cannot depend on y). By Lemma 6, $\vdash pc' \leq \Gamma(y)$. Since $\neg\mathsf{reqErase}(pc', \lfloor\sigma\rfloor_1)$ but $\mathsf{reqErase}(pc', \lfloor\sigma'\rfloor_1)$, there is a variable y' such that $\lfloor\sigma'\rfloor_1(y') \neq \lfloor\sigma\rfloor_1(y')$, and there is an expression $e \in \mathsf{eraseConds}(pc')$ such that y' appears in e, and either $y = y'$ or y can affect whether $\Gamma(y')$ requires erasure, that is (y, y') is in the transitive closure of the overwrite dependency relation $\prec_\Gamma$. From the well-formedness of Γ, we have $\vdash \Gamma(y) \leq \Gamma(y')$, and since $\Gamma \vdash pc'$ pol, $\vdash \Gamma(y') \leq pc'$, and thus $\vdash \Gamma(y) \leq pc'$.

  Since $\sigma'(y)$ is a pair value, by Lemma 12, there is a variable w such that $\sigma'(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau'\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau'\rfloor_2)$ and $\vdash \Gamma(w) \leq \Gamma(y)$. If $\vdash \Gamma(y) \leq \Gamma(w)$, then $\vdash pc' \leq \Gamma(w)$, and so $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(w)$, and so $protected(\Gamma(w), \tau')$. If $\Gamma(y) \not\sqsubseteq \Gamma(w)$, then by Lemma 6, $\sigma(w) = \sigma'(w)$, and so $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_2)$. Because $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $protected(\Gamma(w), \tau)$, and using the inference rules for the extended relabeling relation, we can show that $protected(\Gamma(w), \tau')$. Moreover, by Lemma 5, we have $\tau, \Gamma(w), \Gamma \vdash d_1'$ com and $\tau, \Gamma(w), \Gamma \vdash d_2'$ com. Thus, $\tau', \bot, \Gamma \vdash d'$ com.

  We also need to show that for any variable $z \in Vars$, if $\sigma'(z) = (\!| v_1 \mid v_2 |\!)$ then $protected(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z)$ is a pair value. By Lemma 11, either $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_1)$ or $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$ then $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$. Suppose $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$.

  Suppose $\sigma(z) = \sigma'(z)$. If $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$ we are done. Otherwise $\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$, and so by Lemma 11 and Lemma 12, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_i)$ and $\vdash \Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

  Otherwise, $\sigma(z) \neq \sigma'(z)$ and so $\lfloor\sigma\rfloor_1(z) \neq \lfloor\sigma'\rfloor_1(z)$, and since $\tau, pc', \Gamma \vdash d_1$ com, by Lemma 6 we have $\vdash pc' \leq \Gamma(z)$. Thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_1} \Gamma(z)$ as required.

- OS-PAIR-IF. Here $d =$ **if** e **then** d_0 **else** d_1 and $d' = (\!| d_1' \mid d_2' |\!)$. Since $\sigma(e) = (\!| v_1 \mid v_2 |\!)$, there is at least one variable y that appears in e such that $\sigma(y)$ is a pair value. By Lemma 12, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_2)$ and $\vdash \Gamma(w) \leq \Gamma(y)$ and $\vdash \Gamma(y) \leq p_e$. Therefore, by Lemma 5, we have $\Gamma(w), \Gamma \vdash d_1'$ com and $\Gamma(w), \Gamma \vdash d_2'$ com. Since $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $protected(\Gamma(w), \tau)$, and by Lemma 13, $protected(\Gamma(w), \tau')$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\sigma'\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\sigma'\rfloor_2)$. Thus, $\tau', \bot, \Gamma \vdash d'$ com.

- OS-PAIR-DECLASSIFY. This case is similar to OS-ASSIGN. Here $d = y :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$ and $d' =$ **skip**. By the typing rule for **skip**, we have $\tau', \bot, \Gamma \vdash d'$ com. We need to show that $\forall z \in Vars.\ (\sigma'(z) = (\!| v_1 \mid v_2 |\!)) \Rightarrow protected(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z)$ is a pair value. By Lemma 11, either $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_1)$ or $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if

$\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$ then $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$. Suppose $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$.

Suppose $\sigma(z) = \sigma'(z)$. If $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$ we are done. Otherwise $\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$, and so by Lemma 11 and Lemma 12, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_2)$ and $\vdash \Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

Otherwise, $\sigma(z) \neq \sigma'(z)$ and so either $z = y$, or z was updated by the utility function $erasure(\cdot)$.

Suppose $z = y$. Since $\sigma(e_0 \times \cdots \times e_k)$ is a pair value, there must be a variable w that appears in $e_0 \times \cdots \times e_k$ such that $\sigma(w)$ is a pair, and $\vdash \Gamma(w) \leq \Gamma(z)$ (by the typing rule for declassification). By Lemma 12, there is a variable w' such that $\sigma(w')$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w'), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w'), \lfloor\tau\rfloor_2)$ and $\vdash \Gamma(w') \leq \Gamma(w)$ and $\vdash \Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w')$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

Finally, if $\sigma(z) \neq \sigma'(z)$ and $z \neq y$, then z was updated by the utility function $erasure(\cdot)$. Note that this means $\Gamma(z)$ requires erasure on at least one of the two projections. By Lemma 12, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_2)$ and $\vdash \Gamma(w) \leq \Gamma(z)$. Since $\Gamma(w)$ does not require erasure on either projection, w was not updated by the utility function $erasure(\cdot)$. Thus, by the previous cases, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

∎

Using the type preservation property of IMP^2_E, we are now ready to prove Theorem 2.

Proof of Theorem 2: Let $v_1, v_2 \in \mathbb{Z}$, let σ be a IMP_E memory. Let $\sigma_0 = update^2(\sigma, x, (\!| v_1 \mid v_2 |\!))$. By Lemma 10, $\sigma_0 = erasure^2(\sigma_0)$. Let τ_1 and τ_2 be traces such that $\tau_i[0] = \langle c, \lfloor\sigma_0\rfloor_i\rangle$.

By Lemma 3 and Lemma 1, there is a IMP^2_E trace τ such that $\lfloor\tau\rfloor_i$ is a prefix of τ_i for all $i \in \{1, 2\}$, and for some $i \in \{1, 2\}$, $\lfloor\tau\rfloor_i = \tau_i$.

We construct a correspondence R for τ_1 and τ_2 such that R is the smallest set such that for all $k \in 1..|\tau|$, $(f_1(\tau[..k]), f_2(\tau[..k])) \in R$, where $f_i(\tau')$ is the number of reduction steps of the IMP^2_E execution τ' that reduce the ith projection. Thus, if $(i, j) \in R$, then there is some k such that $\lfloor\tau[k]\rfloor_1 = \tau_1[i]$ and $\lfloor\tau[k]\rfloor_2 = \tau_2[j]$.

Let $\ell \in \mathcal{L}$, and let $(i, j) \in R$. There is some k such that $\lfloor\tau[k]\rfloor_1 = \tau_1[i]$ and $\lfloor\tau[k]\rfloor_2 = \tau_2[j]$. Let $\tau[k] = \langle c_k, \sigma_k\rangle_\bullet$. Note that $\tau_1[i] = \langle\lfloor c_k\rfloor_1, \lfloor\sigma_k\rfloor_1\rangle$ and $\tau_2[j] = \langle\lfloor c_k\rfloor_2, \lfloor\sigma_k\rfloor_2\rangle$.

Suppose for some variable y, $\lfloor\sigma_k\rfloor_1(y) \neq \lfloor\sigma_k\rfloor_2(y)$. Then $\sigma_k(y) = (\!| v_1 \mid v_2 |\!)$. By Theorem 4, we have $\tau[..k], \bot, \Gamma \vdash \langle c_k, \sigma_k\rangle_\bullet$ config. Therefore, $protected(\Gamma(y), \tau[..k])$. Thus, either $\Gamma(x) \leq_{\tau_1[..i]} \Gamma(y)$ or $\Gamma(x) \leq_{\tau_2[..j]} \Gamma(y)$. By Property 2 and Lemma 8, either $(\tau_1[i], \mathsf{obs}(\Gamma(y))) \in [\![\Gamma(x)]\!]_{update(\sigma,x,v_1)}$ or $(\tau_2[j], \mathsf{obs}(\Gamma(y))) \in [\![\Gamma(x)]\!]_{update(\sigma,x,v_2)}$. Therefore, we have if $(\tau_1[i], \ell) \notin [\![\Gamma(x)]\!]_{update(\sigma,x,v_1)}$ and $(\tau_2[j], \ell) \notin [\![\Gamma(x)]\!]_{update(\sigma,x,v_2)}$, then $\tau_1[i] \approx_\ell \tau_2[j]$, and so, c is noninterfering according to policy for variable x. ∎