Required Information Release

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Published Version</td>
<td>doi:10.1109/CSF.2010.22</td>
</tr>
<tr>
<td>Accessed</td>
<td>January 10, 2018 2:18:51 AM EST</td>
</tr>
<tr>
<td>Citable Link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:34401541">http://nrs.harvard.edu/urn-3:HUL.InstRepos:34401541</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>

(Article begins on next page)
Required Information Release

Stephen Chong
TR-04-10

Computer Science Group
Harvard University
Cambridge, Massachusetts
Abstract

Many computer systems have a functional requirement to release information. Such requirements are an important part of a system’s information security requirements. Current information-flow control techniques are able to reason about permitted information flows, but not required information flows.

In this paper, we introduce and explore the specification and enforcement of required information release in a language-based setting. We define semantic security conditions that express both what information a program is required to release, and how an observer is able to learn this information. We also consider the relationship between permitted and required information release, and define bounded release, which provides upper- and lower-bounds on the information a program releases. We show that both required information release and bounded release can be enforced using a security-type system.

1. Introduction

Information-flow control holds the promise of strong, end-to-end, application-specific information security [Sabelfeld and Myers, 2003]. To date, most research on information-flow control has focused on what flows are permitted or prohibited in a system. For example, noninterference [Goguen and Meseguer, 1982] prohibits confidential inputs flowing to public outputs.

Many computer systems release (or declassify) confidential information as part of their intended functionality, and as such, violate noninterference. Much work in recent years has considered weakening noninterference to permit some flow of confidential inputs to public outputs (e.g., Li and Zdancewic [2005]; Chong and Myers [2004]; Clark et al. [2005]; Sabelfeld and Sands [2005]; van der Meyden [2007]; Swamy and Hicks [2008]; Banerjee et al. [2008]).

However, many systems have more than just permission to release information; they have an obligation to release information. In this work, we introduce and explore the specification and enforcement of required information release, or simply, required release.

Examples abound of systems with an obligation to release information. The Sarbanes-Oxley Act of 2002 is a United States federal law that was enacted after a series of corporate accounting scandals, and requires publicly held companies to report details of their finances to a government agency. Thus, financial systems of such companies are required to release sensitive financial information to the government agency. Pharmaceutical companies in many countries are required to report results of clinical trials of new drugs to a government agency (such as the Food and Drug Administration) to receive approval. Computer systems that support the conduct of clinical trials must release all trial results, and not withhold negative results. In general, transparency of organizations and processes requires the release of sensitive information. Other systems that are required to release information include the following.

• Sealed bid auctions: at the end of the auction, the winning bid (and, depending on the auction, the winner’s identity) is required to be released.

• Information purchase: once a customer has paid for information (such as electronic media), the information is required to be available for download.

• Games: legal game-play often requires release of a player’s secret information, such as the cards in a player’s hand, or the location of battleships on a player’s board.

• Course management system: when a professor indicates that exam results are available, the system is required to allow students view their grades.

• Credit card sales: the receipt for a credit card purchase is required to show the final four digits of the customer’s credit card number.

In the examples above, the required release of information is an important aspect of each system’s information security. To gain assurance in the systems’ correct implementation, it is necessary both to specify the required release (and other information security requirements) and to verify that the implementation satisfies the specification.

However, the specification of required release is subtle. What does it mean for a program to satisfy the required release of information? How do we know if a program is successfully and correctly releasing information? It does not suffice for the output of a program to simply depend upon, or be influenced by, the information required for release. Surprisingly, even if the output contains the information required for release, the program may not satisfy the required release of information. We use epistemic logic, and algorithmic knowledge [Halpern et al., 1994] in particular, to guide our definition of required information release. Required information release must specify not just what information is to be released, but also how that information is to be learned by its intended recipient.

Required release is a functional requirement on a system: the output of a program must allow an observer to learn certain information. Required release makes explicit what information the observer learns, and how the observer obtains this knowledge. Noninterference [Goguen and Meseguer, 1982] and most other information flow security conditions are not functional requirements.

However, required release is an information flow security condition; it describes a mandatory flow of information to an observer. By contrast, most existing work in information flow considers permitted flows of information. In terms of dependence, permitted information flow conditions restrict how the output is permitted to


Required Information Release

Stephen Chong
School of Engineering and Applied Sciences
Harvard University

chong@seas.harvard.edu
depend on the inputs. For example, noninterference requires that 
public outputs do not depend on private inputs—if a private input 
changes, the public output should not change. Required informa-
tion release mandates that outputs must depend on inputs in a way 
that allows an observer to learn certain information—if an input 
changes, the output must change to allow an observer to learn the 
specified information.

Required release interacts with permitted information flows in 
more interesting ways than other functional requirements: if a sys-
tem is required to release information, then the system must also 
be permitted to release it. Indeed, required information release and 
permitted information release can be combined to specify both up-
per and lower bounds on the information that a system releases. We 
do so by defining bounded release, a security condition that com-
bines required release and delimited release [Sabelfield and Myers, 
2004], and thus specifies both what information a program is re-
quired to release, and what information it is allowed to release. 

For some systems, bounds on information release should be 
tight. For example, a company producing reports in accordance 
with the Sarbanes-Oxley Act typically wishes to release no more 
information than is required by law; thus, the information that their 
financial system is permitted to release should be identical to the 
information it is required to release. In other systems, the bounds 
are not tight, such as in a poker game where some players are 
permitted, but not required, to reveal their cards at the end of a 
hand.

The remainder of the paper is structured as follows. Section 2 
uses the example of a simple credit card sales system to examine 
what it means for a system to satisfy required information release. 
Section 3 presents an interactive imperative language that we use 
in Section 4 to formally define required release. We also define 
bounded release, a security condition that specifies what informa-
tion a program is required and permitted to release. We show in 
Section 5 that required release and bounded release can be soundly 
enforced in an interactive language by a type system. Section 6 dis-
cusses related work, and Section 7 concludes.

2. What is required release?

Consider, as a running example, a (grossly simplified) credit card 
sales system that takes a credit card number as input from high 
confidentiality channel $H$, and is required to release the last four 
digits to low confidentiality channel $L$ (representing, for example, 
the printer, or an audit log). What does it mean for this system to 
satisfy the required release of the last four digits?

Noninterference [Goguen and Meseguer, 1982] is a strong in-
formation security condition that requires that public outputs reveal 
no information about confidential inputs. Any system that releases 
confidential information violates noninterference; the credit card 
sales system, which must release the last four digits of the confi-
dential credit card number to a publicly observable printer, violates noninterference. However, just because a system violates noninter-
ference does not mean it satisfies the required information release.

Consider the following attempt to implement the credit card 
sales system.

\[
P_1 : \begin{align*} 
\text{input } cc \text{ from } H; \\
\text{if } (cc \mod 10,000) < 5,000 \text{ then } \\
\text{output } 0 \text{ to } L \\
\text{else } \\
\text{output } 1 \text{ to } L 
\end{align*}
\]

The program inputs the credit card number from channel $H$, and 
then outputs either 0 or 1 to channel $L$. The output observed on 
channel $L$ is influenced by the last four digits of the confidential 
input, and thus the program does not satisfy noninterference. How-
ever, the program fails to satisfy the required information release, 
as an observer of channel $L$ does not learn the credit card number’s 
last four digits.

Surprisingly, even if a system outputs the information required 
for release, it may fail to satisfy the required information release. 
This is demonstrated in the following program, which is another 
try to implement the credit card sales system.

\[
P_2 : \begin{align*} 
\text{input } cc \text{ from } H; \\
i := 0; \\
\text{while } i < (cc \mod 10,000) \text{ do } \\
\text{output } i \text{ to } L; \\
i := i + 1; \\
\text{output } (cc \mod 10,000) \text{ to } L; \\
i := i + 1; \\
\text{while } i < 10,000 \text{ do } \\
\text{output } i \text{ to } L; \\
i := i + 1
\end{align*}
\]

The command output $(cc \mod 10,000)$ to $L$ in program $P_2$ above 
explicitly outputs the credit card’s last four digits. However, every 
execution of the program outputs all integers from 0 to 9,999 in 
order. An observer of channel $L$ always sees the same output, 
regardless of the credit card’s last four digits, and so the observer 
learns nothing.

These examples show that it is insufficient for observable output 
to be merely correlated with the information required for release, 
or even for the output to contain that information. The key insight 
is that to satisfy required release, the output must allow an observer 
to know what information was required for release.

In models of knowledge based on possible world semantics 
[Hintikka, 1962; Fagin et al., 1995], an agent has implicit 
knowledge (or, simply, knowledge) of fact $\phi$ if in all possible worlds 
consistent with the agent’s observations, $\phi$ is satisfied. In the credit 
card system, an observer of channel $L$ knows the last four digits 
of the credit card if all credit cards that could have produced the 
observed output end in the same four digits. Programs $P_1$ and $P_2$ 
do not allow an observer of channel $L$ to know the last four digits. 

Standard logical approaches to knowledge suffer from the prob-
lem of logical omniscience: an agent knows all logical conse-
quences of its knowledge. The following attempt to implement 
the credit card system demonstrates this problem. The program 
chooses two large primes, outputs their product, and the result of 
XOR-ing the smaller prime with the last four digits of the credit 
card number (padded with random bits to be the same length as 
the prime).

\[
P_3 : \begin{align*} 
\text{input } cc \text{ from } H; \\
p := \text{generateLargePrime}(); \\
q := \text{generateLargePrime}(); \\
\text{output } p \times q \text{ to } L; \\
\text{if } p < q \text{ then } t := p \text{ else } t := q; \\
\text{output } t \text{ xor pad}(cc \mod 10,000) \text{ to } L
\end{align*}
\]

A logically omniscient observer of the program’s output knows 
what the last four digits of the credit card number are. However, 
determining this requires factoring a large number, which is beyond 
the abilities of humans and current computer systems to perform in 
reasonable time.

**Algorithmic knowledge** [Halpern et al., 1994] was introduced 
to address the problem of logical omniscience, and we can use al-
gorithmic knowledge to reason whether a system satisfies required 
release.

An agent has algorithmic knowledge, or explicit knowledge, of 
fact $\phi$ if the agent has an algorithm that responds “Yes” when 
given input $\phi$ and the agent’s observations. The agent’s knowledge 
algorithm is sound if whenever it responds “Yes” then the agent has
We assume that there is a single channel for each user, which, given the assumption of a single user for each security level, implies a single channel for each security level. We thus identify channels with security levels. An event is the transmission of an input or output on a channel. Event \( in(\ell,v) \) denotes the input of value \( v \) on channel \( \ell \), and event \( out(\ell,v) \) denotes the output of value \( v \) on channel \( \ell \). For simplicity we restrict values to integers.

We use \( Ev_{in} \) and \( Ev_{out} \) to denote, respectively, the set of all input and output events. We use \( Ev(\ell) \) to denote the set of all events that could occur on channel \( \ell \), and \( Ev \) to denote the set of all events.

\[
\begin{align*}
Ev_{in} & \triangleq \bigcup_{\ell \in \mathbb{L}, v \in \mathbb{Z}} \{in(\ell,v)\} \\
Ev_{out} & \triangleq \bigcup_{\ell \in \mathbb{L}, v \in \mathbb{Z}} \{out(\ell,v)\} \\
Ev(\ell) & \triangleq \bigcup_{v \in \mathbb{Z}} \{in(\ell,v), out(\ell,v)\} \\
Ev & \triangleq \bigcup_{\ell \in \mathbb{L}} Ev(\ell)
\end{align*}
\]

Given \( E \subseteq Ev \), an event trace on \( E \) is a finite or infinite sequence \( t = (\alpha_0, \alpha_1, \ldots) \) such that \( \alpha_i \in E \) for all \( i \) such that \( 0 \leq i < |t| \), where \( |t| \) is the length of \( t \). For infinite traces \( t \), we define \( |t| = \infty \). The \( i \)th element of event trace \( t \) is denoted \( t(i) \), for \( i \) such that \( 0 \leq i < |t| \). The empty trace is denoted \( \langle \rangle \). We write \( t t' \) for the concatenation of finite trace \( t \) and trace \( t' \). For traces \( t \) and \( t' \), we say that \( t \) extends \( t' \), written \( t \supseteq t' \), when \( t' \) is a prefix of \( t \). Note that if \( t \) is an infinite trace, then \( t \) is the only trace that extends it. The set of all traces on \( E \) is denoted \( Tr(E) \).

The restriction of trace \( t \) to \( E \), written \( t \upharpoonright E \), is the trace obtained by removing from \( t \) all events not contained in \( E \). We write \( t | \ell \) as shorthand for \( t \upharpoonright Ev(\ell) \).

User strategies express the behavior of users by describing how users interact with a program. Given trace \( t \), a user of a channel with security level \( \ell \) observes the event trace \( t \upharpoonright Ev(\ell) \); a user’s observations may influence their subsequent interaction with the program. Formally, a user strategy for a channel with security level \( \ell \) is a function of type \( Tr(Ev(\ell)) \to \mathbb{Z} \), and expresses what input a user will provide given their previous observations.

Let \( UserStrat \) be the set of all user strategies. A joint strategy is a collection of user strategies, one for each channel. Formally, a joint strategy \( \omega \) is a function of type \( \mathbb{L} \to UserStrat \), that is, a function from security levels to user strategies. Let \( Strat \) be the set of all joint strategies.

User strategies are sensitive information. In general, we want to ensure that lower security users do not learn about strategies employed by higher security users: user \( \ell \) should not learn anything about the strategy of user \( \ell' \), where \( \ell' \not\subseteq \ell \). However, information release will violate this, revealing some information about the strategies of higher security users. In Section 4 we will discuss security requirements, and formally define semantic security conditions.

### 3.2 Syntax and semantics

We use a simple imperative language, extended with input, output, and declassification commands. The syntax of this language is:

\[
\begin{align*}
(\text{expressions}) & \quad e ::= n \mid x \mid e_0 \oplus e_1 \\
(\text{commands}) & \quad c ::= \text{skip} \mid x := e \mid c_0; c_1 \\
& \quad \text{if } e \text{ then } c_0 \text{ else } c_1 \\
& \quad \text{while } e \text{ do } c \\
& \quad \text{input } x \text{ from } \ell \\
& \quad \text{output } c \text{ to } \ell \\
& \quad x := \text{declassify}(e \to \ell)
\end{align*}
\]

Metavariable \( x \) ranges over \( \text{Var} \), the set of all program variables. Variables take integer values, and literal values \( n \) also range over integers. Metavariable \( \oplus \) ranges over total binary operations on the integers. A state \( \sigma \) maps variables to values, and so is a function of type \( \text{Var} \to \mathbb{Z} \). A configuration is a 4-tuple \( (e, \sigma, t, \omega) \).
representing a system about to execute $c$ with state $\sigma$ and joint strategy $\omega$. Finite trace $t$ is the history of events produced by the system so far. Terminal configurations have the form $(\text{skip}, \sigma, t, \omega)$. Metavariable $m$ ranges over configurations.

We define a small-step operational semantics for our language, using the relation $\rightarrow$ over configurations. If $(c, \sigma, t, \omega) \rightarrow (c', \sigma', t', \omega)$ then execution of command $c$ can take a single step to command $c'$, while updating the state from $\sigma$ to $\sigma'$. Trace $t'$ extends $t$ with any events that were produced during the step. Joint strategy $\omega$ is unchanged when a configuration takes a step, and is included in configurations to simplify notation and presentation.

Figure 1 presents inference rules for the operational semantics. We use $\sigma(e)$ to denote the evaluation of expression $e$ where each variable $x$ is replaced with the integer $\sigma(x)$. Input command $\text{input}$ takes the next input value $\ell$ as defined by the user strategy for $\ell$, assigns it to variable $\ell$, and updates the trace with input event $\text{in}(\ell, v)$. Similarly, output command $\text{output}$ emits value $v$ and updates the trace with output event $\text{out}(\ell, v)$. Declassification $x := \text{declassify}(e)$ to semantic level $\ell$ is equivalent to assignment $x := e$; the declassify annotation and security level $\ell$ are used in the type system, described in Section 5.

We use $\rightarrow^*$ to denote the reflexive transitive closure of $\rightarrow$. For finite trace $t$, we say configuration $c$ emits $t$, written $c \rightarrow t$, if there is some configuration $(c, \sigma, t, \omega)$ such that $c \rightarrow^* (c, \sigma, t, \omega)$. For infinite trace $t$, $c$ emits $t$ if $c$ emits all finite prefixes of $t$. Note that emitted events may include both input and output events.

4. Security definitions

In this section we define the security conditions weak required release and strong required release, which formally express what it means for a program to satisfy the required release of information. We also present the security conditions noninterference [Goguen and Meseguer, 1982] and delimited release [Sabelfeld and Myers, 2004]. Noninterference requires that a program does not release any confidential information. Delimited release weakens noninterference by specifying what confidential information a program is allowed to release. We combine delimited release and required release to define bounds on what a program is permitted and required to release.

4.1 Required release

To formally define required release, we must be able to express what information is to be released, and how that information is to be learned by an observer. We introduce input expressions and output expressions to express each of these respectively. Input expressions are expressions over input values supplied on channels; output expressions are expressions over values output on a single channel.

The syntax for input and output expressions is:

\[
\begin{align*}
\text{(input expressions)} & \quad f ::= n \mid f_0 \oplus f_1 \mid \text{in}_i[i] \\
\text{(output expressions)} & \quad g ::= n \mid g_0 \oplus g_1 \mid \text{out}_i[i]
\end{align*}
\]

Input expression $\text{in}_i[i]$ refers to the $i$th input event on channel $\ell$, for $i \in \mathbb{N}$. Input expressions may also contain integer constants and binary operators. Output expressions are evaluated against a trace. The judgment $t \models_\text{in} f \downarrow v$ means that with trace $t$, input expression $f$ evaluates to value $v$. Evaluation rules for input expressions are given in Figure 2. If $t$ does not have an $i$th input event on channel $\ell$, then $\text{in}_i[i]$ evaluates to $\bot$, that is, $t \models_\text{in} \text{in}_i[i] \downarrow \bot$. We assume that any binary operator $\oplus$ defined is total over $\mathbb{Z}_+$ and strict, meaning that for all $m, n \in \mathbb{Z}_+$, $m \oplus n$ is defined, and if $m$ or $n$ is $\bot$, then $m \oplus n = \bot$.

Output expressions are also evaluated against a trace. The judgment $t \models_\text{out} g \downarrow v$ means that output expression $g$ evaluates to value $v$ using trace $t$ restricted to channel $\ell$ events. Output expression $\text{out}_i[i]$ refers to the $i$th output event on channel $\ell$, for $i \in \mathbb{N}$. Figure 2 also presents the evaluation rules for output expressions. Similar to input expressions, if there is no $i$th output event on channel $\ell$, then $\text{out}_i[i]$ evaluates to $\bot$.

Intuitively, user $\ell$ learns input expression $f$ from command $c$ using output expression $g$, if in every execution that $g$ evaluates to an integer value (using the output provided to $\ell$), then $f$ evaluates to the same integer. Thus, input expression $f$ indicates what information the user is to learn, and output expression $g$ indicates how
the user learns it—g provides a sound knowledge algorithm. This leads us to our first definition of required release.

Command c satisfies weak required release of input expression f to user ℓ using output expression g if for any trace t that can be emitted by executing c, if t provides enough output to ℓ to evaluate g, then f and g evaluate to the same value.

**Definition 1** (Weak required release). Command c satisfies weak required release of input expression f to user ℓ using output expression g exactly when:

For all configurations m = (c, σ, (i, ω)) and for all traces t such that \( m \sim t \),

if \( t \models_{out} g \Downarrow v \) and \( v \neq \bot \), then \( t \models_{in} f \Downarrow v \).

Program \( P_4 \) satisfies weak required release of in_{H}[0] mod 10,000 to L using out[1]: the second output to L is the last four digits of the first H input (the credit card number). By contrast, programs \( P_3 \) and \( P_2 \) do not satisfy weak required release of in_{H}[0] mod 10,000 to L for any output expression.

Weak required release is “weak” in that there is no requirement that command c provide sufficient output to ℓ for g to evaluate to an integer value. For example, the program skip satisfies weak required release of any input expression to L using output expression out[0], since no output is ever given to L, and output expression out[0] never evaluates to an integer value.

We can strengthen weak required release to ensure that command c always eventually provides sufficient output to ℓ for g to evaluate to an integer value. Command c satisfies strong required release of input expression f to user ℓ using output expression g if for any trace t that can be emitted by executing c, there is a trace \( t' \) that extends t, can be emitted by executing c, and provides sufficient output to ℓ to evaluate g, and f and g evaluate to the same value.\(^1\)

**Definition 2** (Strong required release). Command c satisfies strong required release of input expression f to user ℓ using output expression g exactly when:

For all configurations m = (c, σ, (i, ω)) and for all traces t such that \( m \sim t \),

there exists trace \( t' \) such that \( t' \supseteq t \), \( m \sim t' \), and

\( t' \models_{out} g \Downarrow v \) and \( t' \models_{in} f \Downarrow v \) for some \( v \neq \bot \).

Strong required release is strictly stronger than weak required release: if command c satisfies strong required release of f to ℓ using g, then it satisfies weak required release of f to ℓ using g. Indeed, weak required release is a form of partial correctness, whereas strong required release is a total correctness condition. The following program satisfies weak required release, but not strong required release of in_{H}[0] mod 10,000 to L using out[1], because in some cases it will never produce sufficient output to L. (For presentation purposes, we assume that constant strings, such as “Last 4 credit card digits: ” can be converted to appropriate constant integer values, and output to channels.)

\[ P_5 : \text{input } cc \text{ from } H; \]

\[ \text{output “Last 4 credit card digits: ” to } L; \]

if cc mod 10 = 0 then (while 1 do skip) else skip;

output (cc mod 10, 000) to L

Program \( P_5 \) satisfies strong required release of in_{H}[0] mod 10,000 to L using out[1], because it always produces appropriate output to channel L.

**Connection to explicit knowledge**

If a program satisfies (weak or strong) required release of input expression f to user ℓ using output expression g, then output expression g provides a sound knowledge algorithm for ℓ to learn the value of f. The knowledge algorithm takes as input a formula \( \phi \) and the sequence of events that ℓ has observed. The knowledge algorithm is straightforward:

If \( \phi \equiv f = n \) and \( t \models_{out} g \Downarrow n \) then respond “Yes”.

Otherwise, respond “No”.

Note that the algorithm never responds “No”, and if the algorithm responds “Yes”, then, because the program satisfies required release of f to ℓ using g, \( t \models_{out} g \Downarrow n \) implies \( f = n \). Thus, the knowledge algorithm is sound.

Strong and weak required release are both parameterized by output expression g. As discussed in Section 2, the output expression g may be specified by the consumer of the output (such as an auditor or government agency), an instruction manual, or may be described by the program’s output (as in Program \( P_5 \), where the text “Last 4 credit card digits” is output just before the last four credit card digits). In practice, there may be additional requirements on the form of the output function, such as a limit on the number of steps required to evaluate it (analogous to requiring that the instructions for a task be no more than two pages).

**Integrity, availability, and properties**

Although required information release is primarily concerned with the confidentiality of information, it is also related to the integrity and availability of information. Weak information release is an integrity requirement: if the output expression evaluates to an integer value, it must equal the evaluation of the input expression. Strong information release contains an availability requirement: the output expression must eventually evaluate to an integer value. Information security requirements are not always easily separable into confidentiality, integrity, and availability requirements.

Weak and strong required release can be defined as properties: predicates over single execution traces. Weak required release is a safety property, and strong required release is neither safety nor

\[ t \models_{in} n \Downarrow n \]

\[ t \models_{in} f_0 \Downarrow v_0 \]

\[ t \models_{in} f_1 \Downarrow v_1 \]

\[ t \models_{out} g_0 \Downarrow v_0 \]

\[ t \models_{out} g_1 \Downarrow v_1 \]

\[ t \models_{out} g_0 \oplus g_1 \Downarrow v \]

\[ t \models_{out} \phi \Downarrow v \]

\[ t \models_{out} \psi \Downarrow \bot \]
liveness [Alpern and Schneider, 1985]. Recent work by Clarkson and Schneider [2008] expresses some information-flow conditions as hyperproperties: predicates of sets of traces. They note that all information-flow conditions they considered were hyperproperties and not properties. Although weak and strong required release are properties, they are clearly constitute part of a systems information flow requirements, so some information-flow conditions of interest are properties. Indeed, in Section 4.3 below, we discuss the relationship between required release and delimited release, an information flow security condition for permitted information release that is a hyperproperty and not a property.

4.2 Noninterference

Noninterference [Goguen and Meseguer, 1982] is a well-known semantic security condition that requires that public observations reveal no secrets. Applied to the interactive setting used here, noninterference ensures that user \( t \) does not gain any knowledge about the strategy employed by any user \( t' \) such that \( t' \not\subseteq t \). That is, the strategy of any such user \( t' \) does not interfere with the trace observed by user \( t \).

More precisely, a program satisfies noninterference if, for all security levels \( \ell \) and all configurations \( m, m' \) that agree on the user strategies of all users \( \ell' \) such that \( \ell' \subseteq \ell \), the traces emitted by \( m \) and \( m' \) are indistinguishable to user \( t \). Two traces \( t \) and \( t' \) are indistinguishable to user \( t \), written \( t \approx t' \) if \( t \parallel \ell \) extends \( t' \parallel \ell \), or vice-versa. This notion of trace indistinguishability is suitable given the observational model, which does not allow a user to directly distinguish a terminated program from a program in a non-terminating loop, or from a program that may eventually produce additional output.

**Definition 3 (Noninterference).** A command \( c \) satisfies noninterference exactly when for all levels \( \ell \in \mathcal{L} \):

For all \( m = (c, \sigma, \ell, \omega) \) and \( m' = (c, \sigma, \ell, \omega') \)

such that \( \omega(\ell') = \omega'(\ell') \) for all \( \ell' \subseteq \ell \),

and for all traces \( t, t' \) such that \( m \Rightarrow t \) and \( m' \Rightarrow t' \),

we have \( t \approx t' \).

Note that both weak and strong required release of \( f \) to \( \ell \) violate noninterference if the input expression \( f \) contains an input expression \( m_n[i] \) such that \( \ell' \not\subseteq \ell \) (and the evaluation of \( f \) depends nontrivially on \( m_n[i] \)). For example, any program that satisfies (weak or strong) required release of \( m_1[0] \) mod 10,000 to \( L \) (such as program \( P_2 \)) must violate noninterference, since \( H \not\subseteq L \), and \( L \) learns something about the strategy of \( H \), to wit, the last four digits of the credit card number that \( H \) entered.

4.3 Delimited and bounded release

Noninterference is a very restrictive security condition. Many real-world programs must violate noninterference in order to satisfy functional requirements that require or allow the release of information.

The security condition delimited release [Sabelfeld and Myers, 2004] weakens noninterference by specifying what information a program is permitted to release.

An escape hatch is a pair \((f, \ell)\) of input expression \( f \) and security level \( \ell \). Intuitively, given escape hatch \((f, \ell)\), a program is permitted to release information \( f \) to security level \( \ell \). Thus, given escape hatches \((f_0, \ell_0), \ldots, (f_n, \ell_n)\), user \( \ell \) is permitted to learn the evaluation of \( f_i \) for any escape hatch \((f_i, \ell_i)\) such that \( \ell_i \not\subseteq \ell \), in addition to the user strategies of any user \( \ell' \) such that \( \ell' \not\subseteq \ell \).

A program satisfies delimited release by escape hatches \((f_0, \ell_0), \ldots, (f_n, \ell_n)\) if, for any security level \( \ell \) and configurations \( m \) and \( m' \) that have the same user strategy for any user \( \ell' \) such that \( \ell' \not\subseteq \ell \), if \( m \) and \( m' \) respectively emit traces \( t \) and \( t' \) that agree on the evaluation of all escape hatches that may release information to level \( \ell \), then the traces emitted by \( m \) and \( m' \) are indistinguishable to user \( \ell \). Formally, we say that traces \( t \) and \( t' \) agree up to \( \ell \) on escape hatches \((f_0, \ell_0), \ldots, (f_n, \ell_n)\) if for all \( i \in \{0, \ldots, n\} \) such that \( \ell_i \not\subseteq \ell \), we have \( t \approx t \) for \( m \) and \( m' \), or \( t \approx t' \) for \( m \) and \( m' \), and for all traces \( t, t' \) such that \( m \Rightarrow t \) and \( m' \Rightarrow t' \), if \( t \) and \( t' \) agree up to \( \ell \) on escape hatches \((f_0, \ell_0), \ldots, (f_n, \ell_n)\), then \( t \approx t' \).

Delimited release generalizes noninterference: if command \( c \) satisfies delimited release by an empty set of escape hatches, then \( c \) satisfies noninterference.

Both delimited release and required release are concerned with information flow, and with the knowledge an observer acquires. Required release specifies what information, at a minimum, a program must release. It specifies what an observer must be able to (explicitly) know, and can be viewed as specifying “lower bounds” on what information a program releases. By contrast, delimited release specifies what an observer is permitted to (implicitly) know, and can be seen as specifying “upper bounds”. The maximum information a program is permitted to release. We can combine the security conditions of delimited release and required release to obtain both upper and lower bounds on a program’s information release.

By analogy with escape hatches (which can be defined as openings in the roof of a building, and specify the upper bounds on information release), we use escape chutes (passages down which things may pass) to define the lower bounds of information release. An escape chute is a tuple \((f, \ell, g)\) of input expression \( f \), security level \( \ell \), and output expression \( g \). We define bounded release by combining delimited release by escape chutes, and required release by escape chutes.

**Definition 5 (Bounded release).** Command \( c \) satisfies weak (strong) bounded release by escape chutes

\((f_0, \ell_0, g_0), \ldots, (f_n, \ell_n, g_n)\)

and escape chutes

\((f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)\)

 exactly when

1. for all \( i \in \{0, \ldots, n\} \), \( c \) satisfies weak (strong) required release of \( f_i \) to \( \ell_i \) using \( g_i \); and
2. \( c \) satisfies delimited release by escape chutes \((f'_0, \ell'_0), \ldots, (f'_n, \ell'_n)\).

Program \( P_2 \) satisfies strong bounded release by escape chute \((m_1[0] \mod 10,000, \ell, \text{out}[1])\) and escape hatch \((m_1[\text{div} 10,000, \ell])\). Thus, bounded release tells us not only that \( P_2 \) releases the input expression \( m_1[0] \mod 10,000 \), but also that this is the only information released by \( P_2 \).

The following program has different upper and lower bounds. It satisfies strong bounded release by escape chute \((m_1[0] \mod 10,000, \ell, \text{out}[1])\) and escape hatch \((m_1[0] \mod 10,000, \ell)\) and \((m_1[0] \div 10^{15}, \ell)\). It always releases the last four digits of the credit card number (via output expression \text{out}[1]) and it may in
addition release information about the first digit of the (16 digit) credit card number.

\[ P_b : \begin{align*}
\text{input } cc & \text{ from } H; \\
& x := \text{declassify}(cc \mod 10.000 \text{ to } L); \\
& \text{output } \text{"Last 4 credit card digits: "} \text{ to } L; \\
& \text{output } x \text{ to } L; \\
& y := \text{declassify}(cc \div 10^{15} \text{ to } L); \\
& \text{if } y = 4 \text{ then output } \text{"Visa" to } L \text{ else skip}
\end{align*} \]

There is a consistency property between the escape hatches and escape chutes. Since escape chutes are the “lower bounds” of information release, they must contain no more information than the escape hatches, the “upper bounds” of information release. More precisely, if \( t \) and \( t' \) are traces that can be produced by a command satisfying bounded release, and \( t \) and \( t' \) agree on all input and output events on all channels up to some level \( \ell \), and agree on the value of all escape hatches that declassify to \( \ell \) or below, then for each escape chute at level \( \ell \) or below, either \( t \) and \( t' \) agree on the value of the escape chute, or \( t \) and \( t' \) do not have sufficient input events to evaluate the escape chute. We say that traces \( t \) and \( t' \) agree on escape chute \((f, \ell, g)\) if \( t \upharpoonright_{in} f \downarrow v_i \) and \( t' \upharpoonright_{in} f \downarrow v_i \) for some \( v_i \neq \bot \).

**Property 1** (Consistency). If command \( c \) satisfies (weak or strong) bounded release by escape chutes \((f_0, \ell_0, g_0), \ldots, (f_n, \ell_n, g_n)\) and escape hatches \((f_0', \ell_0'), \ldots, (f_n', \ell_n')\) then

for all \( \ell \in \mathcal{L} \), and for all configurations \( m = (c, \sigma, \langle \omega \rangle) \) and \( m' = (c, \sigma, \langle \omega' \rangle) \), and for all traces \( t, t' \) such that \( m \sim t \) and \( m' \sim t' \), if \( t \) and \( t' \) agree up to \( \ell \) on escape chutes \((f_0', \ell_0'), \ldots, (f_n', \ell_n')\), and for all \( \ell' \subseteq \ell \) we have \( t |\ell' \equiv t' |\ell' \)

then for all \( i \in \mathbb{N} \) such that \( \ell_i \subseteq \ell \), either \( t \upharpoonright_{in} f_i \downarrow \bot \) or \( t' \upharpoonright_{in} f_i \downarrow \bot \), or \( t \) and \( t' \) agree on escape chute \((f_i, \ell_i, g_i)\).

**Proof. (Sketch)** Given \( \ell \) and \( t \) and \( t' \) such that \( t \) and \( t' \) agree up to \( \ell \) on escape chutes \((f_0', \ell_0'), \ldots, (f_n', \ell_n')\), and \( t \uparrow \ell = t' \uparrow \ell \) for all \( \ell' \subseteq \ell \), then we can construct joint user strategies \( \omega \) and \( \omega' \) such that \( m \sim t \) and \( m' \sim t' \), and \( \omega(\ell') = \omega'(\ell') \) for all \( \ell' \subseteq \ell \).

For any escape chute \((f_i, \ell_i, g_i)\) such that \( \ell_i \subseteq \ell \), suppose \( t_0 \uparrow \ell \downarrow v_i \) and \( t'_0 \uparrow \ell \downarrow v'_i \) for some \( v_i, v'_i \neq \bot \). By delimited release, \( t \equiv t_i \), and so, \( t \) and \( t' \) agree on the values of all output expressions required to evaluate \( g_i \) to an integer value. Therefore, \( v_i = v'_i \). By bounded release, the evaluation of \( f_i \) in \( t \) and \( t' \) also equal \( v_i \), and so \( t \) and \( t' \) agree on escape chute \((f_i, \ell_i, g_i)\).

\[ \square \]

### 5. Enforcement

In this section we show that weak bounded release can be soundly enforced with a security type system. Weak bounded release is the conjunction of weak required release, and delimited release. Since weak required release is a safety property, clearly other enforcement mechanisms could also be used to enforce it, including dynamic mechanisms such as execution monitors. However, due to the similarity of escape chutes and escape hatches, a type system to enforce delimited release can easily be adapted to enforce weak bounded release as well.

Our type system conservatively tracks both the security level of information as it flows through a program, and what input expressions have been output and declassified. This allows us to ensure that (i) confidential information is never output to non-confidential channels; (ii) only appropriate escape hatches are declassified; and (iii) appropriate escape chutes are output to the correct channel in the correct order.

For command \( c \), type judgments have the form

\[ pc, \Gamma; C, \Delta, E, H \vdash c : \Gamma' ; C', \Delta', E', H' \]

where entities to the left of the turnstile (\( \vdash \)) indicate the context before the execution of \( c \), and primed versions on the right hand side of the judgment indicate how the contexts change as a result of the execution of \( c \).

Security level typing context \( \Gamma \) maps variables to security levels, and indicates an upper bound on the information stored in each variable. Program counter level \( pc \) is an upper bound on the information that may cause command \( c \) to be executed, and is used to track implicit information flows [Dennin and Denning, 1977]. Typing context \( \Gamma \) and program counter level \( pc \) are standard in security type systems. Our type system is flow-sensitive, as it allows command \( c \) to modify \( \Gamma \), and is based on the flow-sensitive security type system of Hunt and Sands [2006].

The remaining entities in the context \((C, \Delta, E, H)\) are used to track what input expressions have been output and declassified. Specifically, we conservatively track how many input and output events have been produced on each channel, what input expression (if any) is stored in each variable, what input expressions (if any) have been output, and what input expressions have been declassified.

- \( C : \mathcal{L} \rightarrow \mathbb{Z}_+ \times \mathbb{Z}_+ \) counts the input and output events that have occurred on each channel. If \( C(\ell) = (i, j) \), then the program has received \( i \) input events from channel \( \ell \), and produced \( j \) output events to channel \( \ell \). If \( i = j = \bot \), then an unknown number of input events have been received on channel \( \ell \), and similarly, if \( j = \bot \), an unknown number of output events have been produced.

- \( \Delta : \forall \rightarrow \text{InputExp} \) indicates what input expression is stored in each variable. For any variable \( x \), if \( \Delta(x) = f \) then the value stored in \( x \) is equivalent to input expression \( f \). If \( \Delta(x) = \bot \) then nothing is known about the value stored in \( x \).

- \( E : \mathcal{L} \times \mathbb{Z} \rightarrow \text{InputExp} \) indicates which input expressions have been output to channels. If \( E(\ell, i) = f \) then the \( i \)th value output on channel \( \ell \) is equal to the evaluation of input expression \( f \). If \( E(\ell, i) = \bot \) then either the \( i \)th output to channel \( \ell \) has not yet been produced, or nothing is known about the \( i \)th output to channel \( \ell \).

- \( H : \text{typ}(\text{InputExp} \times \mathcal{L}) \) is a set of escape hatches that may have been declassified.

Figure 3 presents inference rules for the typing judgment. Given a function \( h \), we write \( h[a \mapsto b] \) for the function that evaluates to \( b \) on input \( a \), and otherwise behaves like \( h \). We use \( \Gamma(e) \) to denote an upper bound of all levels \( \Gamma(x) \) for variables \( x \) occurring in \( e \); if \( \mathcal{L} \) is a join semi-lattice, then this is the join of all \( \Gamma(x) \) for \( x \) in \( e \). We extend function \( \Delta \) to a homomorphism on program expressions, and write \( \Delta(e) \) for the result of applying the homomorphism to expression \( e \).

In the typing rules, security level context \( \Gamma \) and program counter level \( pc \) do not interact with other parts of the context, and by themselves form a standard flow-sensitive information-flow security-type system, similar to that of Hunt and Sands [2006]. In the following discussion of the typing rules, we focus on the type system’s novel aspects.

For assignment \( x := e \), T-Assign updates input expression context \( \Delta(x) \) to \( \Delta(e) \), which is either \( \bot \) or an input expression equal to \( e \) at this program point. The typing rule T-Declassify for declassification \( x := \text{declassify}(e \text{ to } \ell) \) is similar to assignment, but escape hatch \( \Delta(e, \ell) \) is added to declassification effect \( H \). Note that the rule implicitly requires \( \Delta(e) \neq \bot \) since \( H \) must
contain escape hatches. Rule T-SEQ simply threads the context through a sequence of commands. A skip command has no effect on the context, shown in rule T-SKIP.

Command input \(x\) from \(\ell\) assigns the next input from channel \(\ell\) to variable \(x\). Rule T-IN updates input expression context \(\Delta\) using helper function \(\text{recordInput}(\Delta, x, C, \ell)\), which updates \(\Delta(x)\) either to \(\bot\) if the number of input events on channel \(\ell\) is not known, or to input expression \(\text{in}_{i}[x]\), where \(i\) is the number of input events received on channel \(\ell\). If known, the number of input events on channel \(\ell\) is incremented using the helper function \(\text{inc}_{in}(C, \ell)\). Command output \(e\) to \(\ell\) outputs expression \(e\) to channel \(\ell\). Using helper function \(\text{recordOutput}(E, C, \ell, f)\), rule T-OUT records that the \(j\)th output on channel \(\ell\) is equal to input expression \(\Delta(e)\), where \(j\) is the number of output events produced on channel \(\ell\), and increments the number of output events produced on channel \(\ell\) with helper function \(\text{inc}_{out}(C, \ell)\). If the number of output events produced on channel \(\ell\) is unknown (i.e., \(j = \bot\)), then no update to \(E\) or \(C\) is made.

The subsumption rule T-SUB allows the context to be weakened, or made less precise. It uses the flat ordering \(\succeq\): for any lifted set \(S_{\bot}\), and for any \(a, b \in S_{\bot}\), \(a \succeq b\) iff \(a = b\) or \(b = \bot\). We extend the \(\succeq\) relation in the obvious way to pairs, and to a pointwise relation over functions. For example, \(\Delta_{0} \succeq \Delta_{1}\) if for all \(x \in \text{Var}\), \(\Delta_{0}(x) \succeq \Delta_{1}(x)\). Similarly, we extend the binary relation \(\sqsubseteq\) over \(L\) to a pointwise relation over functions with codomain \(L\).

The rules for if and while commands (T-IF and T-WHILE respectively) type check their sub-commands with a program counter bound below \(pc\) and \(\Gamma(e)\), since \(e\) controls the execution of the sub-commands. Rule T-WHILE requires that context is unchanged by the execution of the while command; for any channel \(\ell\), this means either that the loop body performs no input or output on \(\ell\), or that the context cannot precisely track the number inputs or outputs received on channel \(\ell\), i.e., \(\Delta(\ell) = (\bot, j)\) and \(\bot \in \{j\}\). Similarly for an if command, the context will lose track of the number of inputs or outputs received on channel \(\ell\) unless both branches always perform the same number of inputs and outputs on \(\ell\).

The type system can easily be converted into an algorithmic type system, using the same technique as Hunt and Sands [2006]. If the security levels \(L\) and binary relation \(\sqsubseteq\) form a join-semi lattice,
then type checking and type inference with the algorithmic type system is decidable in time polynomial in the size of the program.

If command \( c \) is well-typed, then it satisfies both weak required release, and delimited release. Theorem 1 below states this claim formally. To state Theorem 1 concisely, we first introduce a helper function and additional notation.

Helper function \( \text{substOutExp}(E, \ell, g) \) takes output context \( E \), security level \( \ell \), and output expression \( g \), and substitutes any occurrence of \( \text{out}[i] \) with input expression \( E(\ell, i) \), that is, the input expression that \( E \) claims was the \( i \)-th output on channel \( \ell \). For example, if \( E((L, 2)) = \text{in}_H[1] \), then \( \text{substOutExp}(E, L, 42 + \text{out}[2]) = 42 + \text{in}_H[1] \). Rules for \( \text{substOutExp}(E, \ell, g) \) are given in Figure 4.

![Figure 4. substOutExp(E, \ell, g)](image)

We assume there is a notion of equivalence between input expressions, denoted by \( \equiv \). We require that if \( f_0 \equiv f_1 \), then for all traces \( t \) and \( v \in Z_\perp \), \( t \xrightarrow{m} f_0 \Downarrow v \) if \( t \xrightarrow{m} f_1 \Downarrow v \). The equivalence relation could be syntactic identity, or syntactic identity up to commutativity and associativity of operators, or, (depending on the operators in the language) a deeper semantic equivalence.

Finally, for any set \( S \) and \( v \in S \), we use \( \tau \) as shorthand for a constant function that always returns \( v \). For example, \( (0, 0) \) is a function that always returns the pair \( (0, 0) \).

**Theorem 1.** If \( pc, \Gamma_0; (0, 0), \top, \top, \emptyset \vdash c \triangleright \Gamma; C, \Delta, E, H \) for some \( \Gamma_0 \) and \( pc \), then

1. \( c \) satisfies weak required release of input expression \( f \) to user \( \ell \) using output expression \( g \) if \( \text{substOutExp}(E, \ell, g) \equiv f \).
2. \( c \) satisfies delimited release by escape hatches \( (f_0, \ell_0), \ldots, (f_k, \ell_k) \) if for all \( (f, \ell) \in H \) there exists \( i \in 0..k \) such that \( f \equiv f_i \) and \( \ell_i \subseteq \ell \).

A proof of Theorem 1 appears in Appendix A.

If command \( c \) is well-typed, then it satisfies both weak required release, and delimited release, it satisfies weak bounded release.

**Corollary 1.** Command \( c \) satisfies weak bounded release by escape chutes \( (f_0, \ell_0), \ldots, (f_k, \ell_k, g_k) \) and escape hatches \( (f_0, \ell_0), \ldots, (f_k, \ell_k) \) if

\[
\begin{align*}
&pc, \Gamma_0; (0, 0), \top, \top, \emptyset \vdash c \triangleright \Gamma; C, \Delta, E, H \\
&\text{for some } \Gamma_0 \text{ and } pc \text{, and } \text{substOutExp}(E, \ell, g_i) \equiv f_i \text{ for all } i \in 0..k \text{ and for all } (f, \ell) \in H \text{ there exists } i \in 0..n \text{ such that } f \equiv f_i \text{ and } \ell_i \subseteq \ell.
\end{align*}
\]

**Proof.** Immediate from Theorem 1. \( \square \)

Although program \( P_4 \) satisfies bounded release, it does not type-check: it attempts to release information from \( H \) to \( L \) but does not have any declassify annotations. Program \( P_6 \) does type-check. The judgment

\[
L, \top; (0, 0), \top, \top \vdash P_6 \triangleright \Gamma; C, \Delta, E, H
\]

holds for

\[
\Gamma = \top[cc \mapsto H, x \mapsto L, y \mapsto L] \\
C = (0, 0)[H \mapsto (1, 0), L \mapsto (0, \perp)] \\
\Delta = \top[cc \mapsto \text{in}_H[0], x \mapsto \text{in}_H[0] \text{ mod } 10,000, y \mapsto \text{in}_H[0] \text{ div } 10^4] \\
E = \top[(L_0, 0) \mapsto "..", (L_1, 1) \mapsto \text{in}_H[0] \text{ mod } 10,000] \\
H = (\text{in}_H[0] \text{ mod } 10,000, L), (\text{in}_H[0] \text{ div } 10^4, L).}
\]

Thus, by Corollary 1, \( P_6 \) satisfies weak bounded release by escape chute \( (\text{in}_H[0] \text{ mod } 10,000, L, \text{out}[1]) \) and escape hatches \( (\text{in}_H[0] \text{ mod } 10,000, L) \) and \( (\text{in}_H[0] \text{ div } 10^4, L) \).

A more sophisticated static analysis (or a more restrictive language) could enforce strong required release, by reasoning about the termination of loops, and the eventual production of outputs.

### 6. Related work

Much recent work has considered information release. Sabelfeld and Sands [2005] present a survey of work on (permitted) information release, and introduce four aspects of declassification: who, what, where and when. These aspects are also relevant to required information release. This work is primarily concerned with what information is required for release, expressed using input expressions. Strong required release relates to the when aspect: it mandates that information is eventually released, whereas weak required release places no requirements on when (if ever) information is released.

Sabelfeld and Sands [2005] also introduce several prudent principles of declassification. Of these principles, semantic consistency is directly applicable to required release, and is satisfied by weak and strong required release, and bounded release: semantically equivalent programs satisfy the same security conditions. The other principles do not seem directly applicable to required release.

Previous work on specification and enforcement of information release focuses on permitted information release. To the best of our knowledge, this work is the first to address required information release. Appropriate extensions to the policies and enforcement mechanisms of previous work may allow them to reason about required release. The automata for information release due to Swamy and Hicks [2008], and the \texttt{flowspec} used by Banerjee et al. [2008] seem particularly suitable for extending for required release.

Askarov and Sabelfeld [2007b] introduce localized delimited release, which refines delimited release by restricting not only what information may be released, but where it may be released (at an appropriate declassify command). The type system used by Sabelfeld and Myers [2004] to enforce delimited release also enforces localized delimited release, so we speculate that the type system used in this paper to enforce bounded release would also enforce an appropriately defined localized bounded release.

Askarov and Sabelfeld [2007a] define the semantic security condition \textit{gradual release} in terms of attacker knowledge: an attacker’s knowledge of secrets may become more precise only at specified declassification events. Gradual release restricts permitted information release, and as such it suffices to use implicit knowledge; since we are concerned with required release, we use algorithmic knowledge to ensure that knowledge can be obtained with reasonable resources. The use of algorithmic knowledge leads us to specify how an observer learns released information, in addition to what information they learn.

Askarov and Sabelfeld [2009] present semantic security conditions that generalize localized delimited release and gradual release, and consider their enforcement using a combination of static and dynamic techniques in an interactive language. Since weak required release is a safety property, it could be enforced using dynamic techniques, such as execution monitors [Schneider, 2000].
O’Neill [2006] presents many information flow conditions in an epistemic framework, but doesn’t consider algorithmic knowledge or required information release.

As discussed in Section 4.1, required information release is concerned with the integrity and availability of information, in addition to its confidentiality. Zheng and Myers [2005] study the end-to-end enforcement of availability policies, and present a policy framework for specifying confidentiality, integrity, and availability policies. Their policies are based on the decentralized label model [Myers and Liskov, 2000], and specify who may affect the confidentiality, integrity, and availability of information. By contrast, this work focuses on what aspects of availability and integrity: what information must be available, and what it must be equal to.

7. Conclusion

As part of their correct functionality, many systems are required (not just permitted) to release information. This paper introduces the problem of required information release: specifying, reasoning about, and enforcing, the information security of systems that must release information.

We have defined semantic conditions for required information release. Inspired by work on algorithmic knowledge, the semantic conditions must specify both what information is to be released, and how that information is to be learned by an observer. Input expressions specify what information is to be released, and output expressions specify how an observer learns the information. A program satisfies weak required release of input expression $f$ to user $ℓ$ by output expression $g$ if whenever user $ℓ$ is able to evaluate $g$, then $f$ evaluates to the same value. A program satisfies strong required release if it satisfies weak required release, and eventually produces sufficient output for user $ℓ$ to evaluate $g$.

We investigated the relationship between a system’s required and permitted information release, and defined bounded release, which combines required release with delimited release. Bounded release specifies upper and lower bounds on the information a system releases. For many systems, these bounds should be tight: the system should release all and only information it is required to release. We have shown that (weak) bounded release can be conservatively enforced by a type system.

Both weak and strong required release are properties: predicates over single execution traces. Noninterference, delimited release, and many other information security requirements, are hyperproperties, but not properties. One may thus be concerned whether required information release is an information security requirement. We believe that required information release, while a property, is clearly concerned with the flow of information in a system: it requires that, at a minimum, certain information flows to an observer. We have shown a connection between required information release and delimited release: whereas required information release specifies the minimum information flow from high security inputs to low security outputs that a system must satisfy, delimited release specifies maximum information flow. Thus, we believe that required information release is part of a system’s information security requirements.

There is still much left to understand with respect to required information release. There are systems with information release requirements that cannot be expressed using the policies presented in this paper. For example, financial reports of a company should be released to all shareholders, not a subset; if Alice and Bob are the shareholders, the system must release reports to Alice if and only if it releases that information to Bob. In terms of enforcing required information release, it may be impractical to explicitly specify the knowledge algorithm by which an observer may learn the released information; static analyses may allow the automatic discovery of the knowledge algorithm, thus reducing the burden of proving a system satisfies required information release.

To build trustworthy computer systems, it is important to understand and provably enforce a system’s information security requirements. By introducing the concept of required information release, and providing mechanisms to specify and enforce these requirements, this work brings us closer to the goal of strong, end-to-end, application-specific information security.

Acknowledgments

We thank Andrew Myers for very useful discussions and feedback about this work. We also thank Michael Clarkson, Allan Friedman, Tyler Moore, Kevin O’Neill, Fred Schneider, and Jeff Vaughan for interesting and helpful discussions related to this work, and the anonymous reviewers for their useful comments.

References


A. Proof of Theorem 1

In this appendix we prove Theorem 1. Section A.1 shows that a well-typed program satisfies required release, and Section A.2 shows that a well-typed program satisfies delimited release. We first introduce and prove some useful lemmas and theorems about the type system.

We say that a configuration \((c, \sigma, t, \omega)\) satisfies context \(C, \Delta, E\) if the context \(C, \Delta, E\) accurately reflects the configuration. That is, the entity \(C\) records how many input and output events have been received and sent on each channel, and must agree trace \(t\). Similarly, \(\Delta\) records input expressions that are equivalent to values stored in the state, and the state \(\sigma\) and trace \(t\) must agree on these values. Finally, \(E\) records input expressions that are equivalent to output values, and \(t\) must satisfy these relationships.

**Definition 6.** We say that configuration \((c, \sigma, t, \omega)\) satisfies context \(C, \Delta, E, \text{written } C, \Delta, E, \vdash (c, \sigma, t, \omega)\) if all of the following conditions hold.

1. For all \(\ell \in C\), let \(C(\ell) = (i, j)\).
   (a) Either \(i = \bot\) or \(|t| (\text{Ev}_{in} \cap \text{Ev}(\ell))| = i\); and
   (b) Either \(j = \bot\) or \(|t| (\text{Ev}_{out} \cap \text{Ev}(\ell))| = j\).
2. For all \(x \in \text{Var}\), either \(\Delta(x) = \bot\) or \(t \vdash_{in} \Delta(x) \not\models \sigma(x)\).
3. For all \(\ell \in C\) and \(i \in \mathbb{N}\), either \(E(\ell, i) = \bot\) or \(t \vdash_{in} E(\ell, i) \not\models \nu\) where \((t \vdash (\text{Ev}(\ell) \cap \text{Ev}_{out})|(i) = \text{out}(\ell, \nu))\).

If a configuration satisfies context \(C, \Delta, E\), then it also satisfies a weaker, or less precise, context \(C', \Delta', E', \text{etc.}\)

**Lemma 1.** If \(C, \Delta, E, \vdash (c, \sigma, t, \omega)\) and \(C \succeq C'\) and \(\Delta \succeq \Delta'\) and \(E \succeq E'\) then \(C', \Delta', E', \vdash (c, \sigma, t, \omega)\).

**Proof.** For any \(\ell\), let \(C'(\ell) = (i', j')\) and \(C(\ell) = (i, j)\). If \(i' \neq i\) then \(i' = \bot\), and similarly if \(j' \neq j\) then \(j' = \bot\). Therefore condition 1 is satisfied.

Similarly, if \(\Delta(x) \neq \Delta'(x)\), then \(\Delta'(x) = \bot\), and so condition 2 is satisfied.

Finally, for all \(x\) and \(i\), if \(E(\ell, i) \neq E'(\ell, i)\) then \(E'(\ell, i) = \bot\) and so condition 3 is satisfied.

The operational semantics preserve typings.

**Lemma 2 (Type preservation).** If

\[
C, \Delta, E, \vdash (c, \sigma, t, \omega)
\]

and

\[
\text{pc, } \Gamma; C, \Delta, E, H \vdash c > \Gamma'; C', \Delta', E', H'
\]

then there exists \(C'', \Delta'', E'', H'', \Gamma''\) and \(\text{pc'' such that}
\]

\[
\text{pc'', } \Gamma'', C'', \Delta'', E'', H'', \vdash c' > \Gamma''; C', \Delta', E', H'
\]

and

\[
C'', \Delta'', E'', \vdash (c', \sigma', t', \omega).
\]

**Proof.** By induction on the typing judgment \(\text{pc, } \Gamma; C, \Delta, E, H \vdash c > \Gamma'; C', \Delta', E', H'\).

- **T-Skip**
  Trivial, as there is no \((c', \sigma', t', \omega)\) such that \((\text{skip}, \sigma, t, \omega) \rightarrow (c', \sigma', t', \omega).

- **T-Assign**
  In this case, \(c = x := c', \sigma' = \sigma(x \mapsto \sigma(e))\), \(t' = t, C' = C, \Delta' = \Delta[x \mapsto \Delta(e)], E' = E, H' = H, \Gamma' = \Gamma[x \mapsto \ell]\).
  Let \(C'' = C', \Delta'' = \Delta', E'' = E', H'' = H', \Gamma'' = \Gamma'\).
  Then \(\text{pc, } \Gamma'', C'', \Delta'', E'', H'', c' > \Gamma''; C', \Delta', E', H'\).
  Since \(C, \Delta, E, \vdash (c, \sigma, t, \omega)\) we have for all \(y \in \text{Var}\) such that \(y \neq x\) either \(\Delta''(y) = \bot\) or \(t \vdash_{in} \Delta''(y) \not\models \sigma(y)\). We have \(\Delta''(x) = \Delta(e)\), and if \(\Delta(x) \neq \bot\), then, since \(\sigma''(x) = \sigma(x)\), we can show by induction on the structure of \(c\) that \(t \vdash_{in} \Delta''(x) \not\models \sigma(x)\). Moreover, since \(C'' = C\) and \(E'' = E\) and \(t' = t\) we have \(C'', \Delta'', E'', \vdash (c', \sigma', t', \omega)\) as required.

- **T-Seq**
  In this case, \(c = c_0\); \(C_1\). If \(c_0 \neq \text{skip}\) then the result follows trivially from the inductive hypothesis. If \(c_0 = \text{skip}\) then \(c' = c_1\), and \(\sigma' = \sigma\) and \(t' = t\). The result follows easily from \(\text{pc, } \Gamma; C, \Delta, E, H \vdash c' > \Gamma''; C', \Delta', E', H'\).

- **T-Input**
  In this case, \(c = \text{input } \ell\) from \(x, c' = \text{skip}, \sigma' = \sigma[x \mapsto v]\), \(t' = t'\langle \text{in}(\ell, v)\rangle\), \(C' = \text{inc}_{in}(C, \ell), \Delta' = \text{recordInput}(\Delta, x, C, \ell), E' = E, H' = H, \Gamma' = \Gamma[x \mapsto \ell]\).
  Let \(C'' = C', \Delta'' = \Delta', E'' = E', H'' = H', \Gamma'' = \Gamma'\).
  Then \(\text{pc, } \Gamma'', C'', \Delta'', E'', H'', c' > \Gamma''; C', \Delta', E', H'\).
  Also, either \(j = \bot\) or \(|t'| (\text{Ev}_{out} \cap \text{Ev}(\ell))| = j\).
  If \(\Delta''(x) \neq \bot\) then \(i = \bot\) and \(\Delta''(x) = \text{in}_{in}[i]\). Since \(\sigma'(x) = v\) and \(t' \not\models (\text{Ev}(\ell) \cap \text{Ev}_{in}(\ell))\)(i) = \text{in}(\ell, v)\) we have \(t' \vdash_{in} \Delta(x) \not\models \sigma'(x)\).
  Thus we have \(C'', \Delta'', E'', \vdash (c', \sigma', t', \omega)\) as required.
Lemma 3.
Proof by induction on $\ell$ and $k$. Given $\ell$, $k$, and $C(\ell, k)$, we have $C' = \text{inc}_{out}(C, \ell, \Delta(\ell))$, $H' = H'$, $\Gamma' = \Gamma'$. Let $C'' = C'$, $\Delta'' = \Delta'$, $E'' = E'$, $H'' = H'$, $\Gamma'' = \Gamma'$. Then $pc, \Gamma'', \Delta'', E'', H'' \vdash c \triangleright \Gamma', C', \Delta', E'$, $H'$ as required. If $C'' \not\subseteq C$ then $C''(\ell) = (i, j)$ for some $i$, $j$ and $C(\ell) = (i, j-1)$. Since $|t| \subseteq \text{Eout}(\ell) = \text{Eout}(\ell)$ where $\text{Eout}(\ell) = \text{Eout}(\ell)$, we have $|t' | \subseteq \text{Eout}(\ell) = \text{Eout}(\ell)$. Also, either $i \not\subseteq \bot$ or $j \not\subseteq \bot$.

Similarly, the type system ensures that for typing context $E$ (which records input expressions that are equivalent to output values), the post-context may be less precise than the pre-context, but otherwise agrees with it. That is, once the type system has recorded that a given expression is equivalent to a given output value, the type system can not change it to a different input expression.

**Lemma 4.** Given $pc, \Gamma; C, \Delta, E, H \vdash c \triangleright \Gamma'; C', \Delta', E', H'$ and $\ell \in L$, let $C(\ell) = (i, j)$ and $C'(\ell) = (i', j')$. Either $i \not\subseteq \bot$ and $i' \not\subseteq \bot$ and $i < i'$ or $i' < i$. Also, either $j \not\subseteq \bot$ and $j' \not\subseteq \bot$ and $j < j'$ or $j' < j$.

**Proof.** By induction on the typing judgment $pc, \Gamma; C, \Delta, E, H \vdash c \triangleright \Gamma'; C', \Delta', E', H'$.

- **T-Skip, T-Assign, T-While, T-Declassify**
  Trivial, as $C' = C$.
- **T-Seq, T-If**
  Follows easily from the inductive hypothesis.
- **T-Sub**
  Follows easily from the inductive hypothesis, and ordering $\succeq$.
- **T-In**
  Here $C'$ is identical to $C$ with one possible exception for some $\ell'$, in which case $C''(\ell') = (i+1, j)$ where $C(\ell') = (i, j)$.
- **T-Out**
  Here $C'$ is identical to $C$ with one possible exception for some $\ell'$, in which case $C''(\ell') = (i+1, j)$ where $C(\ell') = (i, j)$.
Lemma 6. If $C$, and $\ell$ then for all pc, $E \vdash c \triangleright C''$, $\Delta''$, $E''$, $H''$ and $pc''$, $\Gamma''$, $C''$, $\Delta''$, $E''$, $H'' \vdash c_1 \triangleright C'$, $\Delta'$, $E'$, $H'$.

If $j \leq \perp$ then by the inductive hypothesis, we have for all $k \in \mathbb{N}$, we have $E(\ell, k) \geq E''(\ell, k)$. Moreover, by Lemma 3, $C''(\ell) = (\perp', \perp)$ for some $\perp'$. Thus, by another application of the inductive hypothesis, we have for all $k \in \mathbb{N}$, we have $E''(\ell, k) \geq E'(\ell, k)$, and so for all $k \in \mathbb{N}$, we have $E(\ell, k) \geq E'(\ell, k)$ as required.

If $j \leq \perp$ then by the inductive hypothesis, for all $0 \leq k < j$, we have $E(\ell, k) \geq E''(\ell, k)$. By Lemma 3, $C''(\ell) = (\perp', j')$ and either $j' > j$ or $j' = \perp$. Either way, by another application of the inductive hypothesis for all $0 \leq k < j$, we have $E''(\ell, k) \geq E'(\ell, k)$, and so for all $0 \leq k < j$, we have $E(\ell, k) \geq E'(\ell, k)$ as required.

T-SUB

Here we have $\text{pc}_1, \Gamma_1; \Delta_1, E_1, H_1 \vdash c \triangleright \Gamma_1'; C'_1, \Delta_1', E'_1, H'_1$ where $E \geq E_1$ and $E'_1 \geq E'$.

Proof. By induction on $\text{subOutExp}(E, \ell, g)$. The only interesting case is $g = \text{out}[i]$ (where $\text{out}[i]$ may affect the evaluation of the whole output expression). In that case, $\text{subOutExp}(E, \ell, g) = E(\ell, i) \neq \perp$. We have $(t \triangleright (E_{\text{out}} \cap \text{Ev}(\ell)))[i] = \text{out}(\ell, v)$, and so, since $C, \Delta, E, \vdash (c, \sigma, t, \omega)$, we have $t \vdash_{\text{in}} E(\ell, i) \downarrow v$.

Another useful lemma is that the program counter level pc is a lower bound on the side effects of a command.

Lemma 6. If 

$$pc, \Gamma; C, \Delta, E, H \vdash c \triangleright C', \Delta', E', H'$$

and

$$(c, \sigma, t, \omega) \rightarrow (c', \sigma', t', \omega),$$

then for all $\ell$ such that $pc \not\subseteq \ell$, we have

- $t \mid \ell' = t' \mid \ell$ and
- $t \vdash (c') (\ell') = t \vdash (c) (\ell)$.

Proof. By induction on $pc, \Gamma; C, \Delta, E, H \vdash c \triangleright C', \Delta', E', H'$. The interesting cases are T-ASSIGN, T-DECLASSIFY, T-IN, and T-OUT.

In each of these cases $pc$ is a lower bound on the side effect.

A.1 Required release

Having shown several useful lemmas, we are now ready to prove that well-typed programs satisfy required release.

Lemma 7. If 

$$pc, \Gamma_0; (0, 0), \top, \top, \emptyset \vdash c \triangleright \Gamma; C, \Delta, E, H$$

for some $\Gamma_0$ and pc, and $\text{subOutExp}(E, \ell, g) \equiv f$, then $c$ satisfies weak required release of input expression $f$ to user $\ell$ using output expression $g$.

Proof. Assume $pc, \Gamma_0; (0, 0), \top, \top, \emptyset \vdash c \triangleright \Gamma; C, \Delta, E, H$ for some $\Gamma_0$ and pc, and $\text{subOutExp}(E, \ell, g) \equiv f$. Let $m = (c, \sigma, (\ell), \omega)$ for some $\sigma$ and $\omega$. Suppose that $m \rightarrow (c', \sigma', t, \omega)$ and $t \vdash_{\text{out}} f \downarrow v$ for some $v \neq \perp$. We need to show that $t \vdash_{\text{in}} f \downarrow v$.

By repeated application of Lemma 2, there exists $C', \Delta', E', H', \Gamma'$, and $pc''$ such that $pc', \Gamma'; C', \Delta', E', H' \vdash c' \triangleright \Gamma; C, \Delta, E, H$ and $C', \Delta', E', \vdash (c', \sigma', t, \omega)$. Since $t \vdash_{\text{out}} f \downarrow v$ for $v \neq \perp$, there have been sufficient output events to channel $\ell$ to evaluate $g$ to a non-$\perp$ value. Since $C', \Delta', E', \vdash (c', \sigma', t, \omega)$, we have $C'(\ell) = (i, j)$ and either $j = \perp$ or $j = \perp' \mid (\text{Ev}_{\text{out}} \cap \text{Ev}(\ell))$. Either way, by Lemma 4, we have $\text{subOutExp}(E', \ell, g) = \text{subOutExp}(E, \ell, g)$. Since $\text{subOutExp}(E', \ell, g) \equiv f$, then $E'(\ell, i) \neq \perp$ for all $i$ such that $\text{out}(\ell, i)$ appears in $g$ (that may affect the evaluation of $g$). By Lemma 5, $t \vdash_{\text{in}} \text{subOutExp}(E', \ell, g) \downarrow v$. From the definition of $\equiv$, we have $t \vdash_{\text{in}} f \downarrow v$ as required.

A.2 Delimited release

We prove that the type system enforces delimited release user a proof technique based on the technique of Pottier and Simonet [2002] for showing noninterference in the ML programming language. We define a new language, IMP2, that can represent two executions of a program. We show that type preservation in IMP2 implies that the program satisfies delimited release. (For convenience, we use IMP to refer to the interactive imperative language presented in Section 3.)
A.2.1 Syntax and semantics

The language IMPL allows the use of pair construct for commands \( \langle c_1, c_2 \rangle \), integers \( \langle v_1, v_2 \rangle \), and events \( \langle \alpha_1, \alpha_2 \rangle \).

The pair construct represents different commands, integers, and events that may arise in 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 represent different integer values that may be provided by different user strategies, and to track how states differ in different executions of a program:

user strategies in IMPL are functions from IMPL traces to integers and integer pairs, and stores in IMPL are functions from variables to integers and integer pairs. We introduce the special event void, and allow elements of event pairs to range over events and void. The constant void is used to indicate that an event occurred in only one of the two executions. We also allow integer and output values to range over integer pairs and integers.

For an extended command \( c \), let the projections \( \langle c_1 \rangle \) and \( \langle c_2 \rangle \) represent the two commands that \( c \) encodes. The projection functions satisfy \( \langle v_1, v_2 \rangle = v_i \). The projection functions are extended to states, so that

\[
[s](x) = \begin{cases} v & \text{if } s(x) = n \\ v_1 & \text{if } s(x) = \langle v_1, v_2 \rangle \end{cases}
\]

The evaluation of expressions is also extended, so that binary operations \( \oplus \) are homomorphically on integer pairs. Thus, \( e(c) \), the evaluation of expression \( e \) using state \( s \), may be either an integer \( v \) or an integer pair \( \langle v_1, v_2 \rangle \).

We extend projection to event pairs \( \langle \langle \alpha_1, \alpha_2 \rangle \rangle \) and define projection homomorphically on events. We defineprojection on traces inductively, as follows.

\[
[\langle \rangle]_i = \langle \rangle \\
[\langle \alpha_0, \alpha_1, \ldots \rangle]_i = \begin{cases} [\langle \alpha_1, \ldots \rangle]_i & \text{if } [\alpha_0]_i = \emptyset \\ [\langle \alpha_0 \rangle]_i[\langle \alpha_1, \ldots \rangle]_i & \text{otherwise} \end{cases}
\]

Finally, we extend projection to joint strategies, so that for any \( \ell \) and \( t \), \([\omega_i](\ell)(t) = [\omega](\ell)(t)\).

We indicate IMPL configurations with a bullet (●) subscript: \( c, \sigma, t, \omega \). A IMPL configuration represents a pair of IMPL configurations.

The complete operational semantics of IMPL are given in Figure 5. Note that rules OS-i-ASSIGN, OS-i-SEQ-i, OS-i-SEQ-2, OS-i-IN, OS-i-OUT, OS-i-IF-i, OS-i-IF-2, OS-i-DECLASSIFY, and OS-i-WHILE are similar to their counterparts in the language IMPL. Rules OS-i-IF-i and OS-i-DECLASSIFY have been modified to restricted to only to integer results of evaluating expression \( e \).

The rule OS-PAIR-LIFT evaluates one of the two subcommands of a command pair \( \langle c_1, c_2 \rangle \). The memory and trace are update to indicate that only one of the two configurations represented by the configuration made progress. Thus, \([\sigma]_j = [\sigma']_j \) and \([t]_j = [t']_j \), where \( j \in \{1, 2\} \) is the execution that did not make progress. Note that the small step relation used in the premise is small step relation of language IMPL.

The rule OS-PAIR-SKIP applies when the two commands represented by a command pair have both finished executing. This rule removes the command pair.

The rule OS-PAIR-IF applies when the evaluation of a conditional expression differs in the two executions represented by the IMPL configuration. This rule introduces a command pair, representing the different branches that may be taken by the two executions. This is the only rule that introduces command pairs.

The rule OS-PAIR-DECLASSIFY applies when an expression is declassified in both executions represented by the execution, and the evaluation of the expression is the same in both executions.

A.2.2 Adequacy

The language IMPL is adequate to express the execution of two IMPL programs. We show that the execution of a IMPL program is sound (a step taken by a IMPL program corresponds to one or zero steps taken by its projections) and complete (given two IMPL executions, there is a IMPL execution whose projection agrees with at least one of them). We write \( \longrightarrow^* \) for the reflexive closure of \( \rightarrow \).

Lemma 8 (Soundness). If \( (c, \sigma, t, \omega) \) \( \longrightarrow (c', \sigma', t', \omega) \), then \( ([c_i], [\sigma_i], [t_i], [\omega_i]) \longrightarrow^* ([c'_i], [\sigma'_i], [t'_i], [\omega_i]) \) for \( i \in \{1, 2\} \).

Proof. By induction on the derivation \( (c, \sigma, t, \omega) \) \( \rightarrow (c', \sigma', t', \omega) \). The interesting cases are the new rules introduced for IMPL: 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.

If an IMPL configuration is stuck, it is because one of the two projections is stuck.

Lemma 9 (Stuck configurations). If \( (c, \sigma, t, \omega) \) is stuck (i.e., cannot be reduced and \( c \neq \text{skip} \)), then \( ([c_i], [\sigma_i], [t_i], [\omega_i]) \) is stuck for some \( i \in \{1, 2\} \).

Proof. By structural induction on command \( c \).

Given two IMPL evaluations, there is an IMPL evaluation that represents the same IMPL evaluation for at least one of the two evaluation.

Lemma 10 (Completeness). If \( ([c_i], [\sigma_i], [t_i], [\omega_i]) \rightarrow^* ([c'_i], [\sigma'_i], [t'_i], [\omega_i]) \) for \( i \in \{1, 2\} \), then there exists an IMPL configuration \( (c', \sigma', t', \omega) \) such that \( (c, \sigma, t, \omega) \) \( \rightarrow^* (c', \sigma', t', \omega) \) and \( ([c'_i], [\sigma'_i], [t'_i], [\omega_i]) = ([c_i], [\sigma_i], [t_i], [\omega_i]) \) for some \( i \in \{1, 2\} \).
\textbf{OS}^2-\text{ASSIGN} \quad (x := e, \sigma, t, \omega) \xrightarrow{} (\text{skip}, \sigma[x \mapsto \sigma(e)], t, \omega)

\textbf{OS}^2-\text{SEQ-1} \quad (\text{skip}; e, \sigma, t, \omega) \xrightarrow{} (e, \sigma, t, \omega)

\textbf{OS}^2-\text{SEQ-2} \quad (e_0; e, \sigma, t, \omega) \xrightarrow{} (e'_0, \sigma', t', \omega)

\textbf{OS}^2-\text{OUT} \quad \sigma(e) = v

\textbf{OS}^2-\text{IF-1} \quad \sigma(e) \neq 0 \quad \sigma(e) \in \mathbb{Z}

\textbf{OS}^2-\text{IF-2} \quad (\text{if } e \text{ then } e_0; e_1, \sigma, t, \omega) \xrightarrow{} (e_0, \sigma, t, \omega)

\textbf{OS}^2-\text{WHILE} \quad (\text{while } e \text{ do } e, \sigma, t, \omega) \xrightarrow{} (\text{if } e \text{ then } (e; \text{while } e \text{ do } e) \text{ else } \text{skip}, \sigma, t, \omega)

\textbf{OS}^2-\text{PAIR-LIFT} \quad \{i, j\} = \{1, 2\}

\begin{align*}
\sigma' &= \lambda x. \begin{cases} 
\{\sigma'_1(x) | \sigma'_2(x)\} & \text{if } \sigma_i(x) \neq \sigma'_i(x) \\
\sigma(x) & \text{otherwise}
\end{cases} \\
t' &= \begin{cases} 
t & \text{if } [t]_i = t'_i \\
t'(\ell \alpha_0 | \alpha_1) & \text{if } [t]_i \neq t'_i
\end{cases} \\
\alpha_i &= t'_i([t'_i] - 1) \\
\alpha_j &= \text{void}
\end{align*}

\textbf{OS}^2-\text{IF} \quad \sigma(e) = \langle v | v \rangle

\textbf{OS}^2-\text{PAIR-DECLASSIFY} \quad (x := \text{declassify}(e \text{ to } t), \sigma, t, \omega) \xrightarrow{} (\text{skip}, \sigma[x \mapsto \sigma(e)], t, \omega)

\textbf{OS}^2-\text{PAIR-SKIP} \quad (\langle \text{skip} | \text{skip} \rangle, \sigma, t, \omega) \xrightarrow{} (\text{skip}, \sigma, t, \omega)

Figure 5. Operational semantics of IMPI^2

\textbf{Proof.} Let $\tau_i = ([c], \{\sigma_i, \{[t]_i, \omega_i\}) \ldots \{c'_i, \sigma'_i, t'_i, \omega_i\})$ be the sequence of configurations that witnesses $([c], \{\sigma, \{[t], \omega\}) \xrightarrow{}^\ast (c', \sigma', t', \omega))$

Let $n_i$ be the length of $\tau_i$. For a sequence IMPI^2 configurations $\tau = (c, \sigma, t, \omega), \ldots (c', \sigma', t', \omega)$, that witnesses $([c], \{\sigma, \{[t], \omega\}) \xrightarrow{}^\ast (c', \sigma', t', \omega))$, let $f_1(\tau)$ be $n_i$ minus the number of reduction steps in $\tau$ that reduce the $i$th projection. Note that $f_1(\tau)$ is non-negative. Consider $g(\tau) = \min\{f_1(\tau), f_2(\tau)\}$. If $g(\tau) = 0$, then $\tau$ is a sequence that satisfies the conditions.

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

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

where $\tau[|\tau| - 1]$ refers to the last configuration in the sequence $\tau$, and $\text{numPairs}(\langle c, \sigma, t, \omega \rangle)$ 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 $\tau$ by one step to a sequence $\tau'$ such that $h(\tau') < h(\tau)$ under lexicographic ordering, then, by repeated applications, eventually we will produce a sequence $\tau''$ such that $g(\tau'') = 0$.

We now show how to extend sequence $\tau$ to a sequence $\tau'$ such that $h(\tau') < h(\tau)$. By assumption, $g(\tau) > 0$, so neither last configuration of $\tau_1$ or $\tau_2$ is stuck. By Lemma 9, we can extend $\tau$ by one more step, producing trace $\tau'$. By Lemma 8, either $f_1(\tau') = f_1(\tau) - 1$ for some $i \in \{1, 2\}$, or $f_2(\tau') = f_2(\tau)$ for all $i \in \{1, 2\}$. If the former, then $h(\tau') < h(\tau)$. If the latter, then the rule OS\textsuperscript{2}-PAIR-SKIP was used in the reduction, and the last configuration of $\tau'$ has one fewer pair commands than the last configuration of $\tau$, and so $h(\tau') < h(\tau)$.

\[ \square \]
A.2.3 Type preservation

We extend the type system to IMPl\(^2\) commands and configurations. Type judgments now are of the form

\[ pc, \Gamma; C, \Delta, E, H \vdash_\ell c \triangleright \Gamma'; C', \Delta', E', H' \]

where \(\ell \in \mathcal{L}\). Intuitively, if IMPl\(^2\) command \(c\) is well-typed with typing parameter \(\ell\), then \(c\) represents the two IMPl commands that are indistinguishable from the point of view of any user \(\ell'\) such that \(\ell' \subseteq \ell\).

Because our type system is flow-dependent, we need to extend the typing context entities \(\Gamma, \Delta,\) and \(E\) so that they range over pairs. This allows the expression of different typing contexts for the two IMPl commands represented by a single IMPl\(^2\) command. Thus, \(\Gamma\) ranges over security levels \(\mathcal{L}\) and pairs of security levels (written \(\langle \ell_1, \ell_2 \rangle\)). Similarly, \(\Delta\) and \(E\) range over elements of IMPlExp\(_\downarrow\) and pairs of elements of IMPlExp\(_\downarrow\). We also extend the entity \(C\) so that its range is \((\mathcal{L} \cup \mathcal{L} \times \mathcal{L}) \times (\mathcal{L} \cup (\mathcal{L} \times \mathcal{L}))\). That is, for any \(\ell \in \mathcal{L}\), we have \(\Gamma(\ell)(i, j)\), where \(i\) is either an integer (indicating the number of inputs received on channel \(\ell\)), \(\downarrow\) (indicating an unknown number of inputs received on channel \(\ell\)), or a pair \((i_1, i_2)\), where \(i_1\) indicates inputs received on channel \(\ell\) in the first execution, and \(i_2\) indicates inputs received on channel \(\ell\) in the second execution. Similarly, \(j\) describes the outputs sent on channel \(\ell\) in both executions. We define projection operations \([\cdot ]_1\) and \([\cdot ]_2\) for all the extended entities. We extend relations \(\subseteq\) and \(\supseteq\) such that

\[ \ell \subseteq \{ \ell_1, \ell_2 \} \iff \ell \subseteq \ell_1 \text{ and } \ell \subseteq \ell_2 \]

\[ \langle \ell_1, \ell_2 \rangle \subseteq \ell \iff \ell_1 \subseteq \ell' \text{ and } \ell_2 \subseteq \ell' \]

\[ \langle v_1, v_2 \rangle \supseteq v \iff v_1 \supseteq v_1 \text{ and } v_2 \supseteq v_2 \]

Typing rules for IMPl\(^2\) (given in Figure 3) are made into typing rules for IMPl\(^2\) by adding the typing parameter \(\ell\) to every rule. In addition, we severely restrict when the typing context entities may differ for the two different IMPl commands represented by a single IMPl\(^2\) command. We require for judgment \(pc, \Gamma; C, \Delta, E, H \vdash_\ell c \triangleright \Gamma'; C', \Delta', E', H'\) that (a) the image of \(C', \Delta', E'\) and \(\Gamma'\) do not contain any pairs (i.e., they are suitable IMPl entities); and (b) if \(c\) does not contain a command pair, then \(C, \Delta, E\) and \(\Gamma\) do not contain any pairs. We use the predicate noPairs\((\Gamma, C, \Delta, E)\) to indicate that the image of \(C, \Delta, E\) and \(\Gamma\) do not contain any pairs.

All typing rules for IMPl\(^2\) are presented in Figure 6. The typing rule for the new pair command, T-PAIR, requires that both projections type check using IMPl typing rules, for a program counter level \(pc\) that is at least as restrictive as typing parameter \(\ell\). Intuitively, this will ensure that any side-effects of a command pair will not be observable at level \(\ell\) or below. Note that the premise of T-PAIR uses the typing judgment for IMPl, i.e., without the typing parameter \(\ell\). This is because well-formed commands do not have nested command pairs. All typing rules other than T-PAIR correspond closely to their IMPl counterpart.

We define a notion of satisfaction for IMPl\(^2\) configurations. Intuitively, an IMPl\(^2\) configuration satisfies context \(C, \Delta, E, \Gamma\) for \(\ell\) if the two IMPl configurations represented by the IMPl\(^2\) configuration are identical at all levels \(\ell'\) such that \(\ell' \subseteq \ell\), and each IMPl configuration satisfies the appropriate IMPl context. We also require that no command pair appears as a subcommand of an if or while command.

**Definition 7.** We say that configuration \((c, \sigma, t, \omega)\), satisfies context \(C, \Delta, E, \Gamma\) for \(\ell\), written \(\Gamma, C, \Delta, E \vdash_\ell (c, \sigma, t, \omega)\), if all of the following conditions hold.

1. For all \(x \in \text{Var.}\) if \(\sigma(x)\) is a pair value then \(\Gamma(x) \not\subseteq \ell\).
2. For all \(i\) such that \(0 \leq i < |t|\), if value\((t(i))\) is a pair value, then level\((t(i))\) \(\not\subseteq \ell\), where

\[ \text{value}((\alpha_1, \alpha_2)) = \{0, 0\} \]

\[ \text{value}(\text{null}(\ell, v)) = v \]

\[ \text{value}(\text{out}(\ell, v)) = v \]

\[ \text{level}(\langle \text{void}, \alpha \rangle) = \text{level}(\alpha) \]

\[ \text{level}(\langle \alpha, \text{void} \rangle) = \text{level}(\alpha) \]

\[ \text{level}(\text{in}(\ell, v)) = \ell \]

\[ \text{level}(\text{out}(\ell, v)) = \ell \]

3. For all \(\ell', \ell''\), if \(\omega(\ell')(\ell'')\) is a pair value, then \(\ell'' \not\subseteq \ell\).
4. \([\mathcal{C}]_i, [\mathcal{E}]_i, [\mathcal{I}]_i, [\mathcal{J}]_i, [\mathcal{S}]_i, [\sigma]_i, [t]_i, [\omega]_i\) for \(i \in \{1, 2\}\).
5. No command pair appears as a subcommand of an if or while command of \(c\).

If a configuration satisfies context \(C, \Delta, E, \Gamma\) for \(\ell\), then it also satisfies a weaker, or less precise, context \(C', \Delta', E', \Gamma'\).

**Lemma 11.** If \(\Gamma, C, \Delta, E \vdash_\ell (c, \sigma, t, \omega)\), and \(C \supseteq C'\) and \(\Delta \supseteq \Delta'\) and \(E \supseteq E'\) and \(\Gamma \supseteq \Gamma'\) then \(\Gamma', C', \Delta', E' \vdash_\ell (c, \sigma, t, \omega)\).

**Proof.** If \(\Gamma(x)\) is a pair value, then \(\Gamma(x) \not\subseteq \ell\), and \(\Gamma(x) \not\subseteq \ell\), then \(\Gamma(x) \not\subseteq \ell\), a contradiction.

We have \([\mathcal{C}]_i, [\mathcal{E}]_i, [\mathcal{I}]_i, [\mathcal{J}]_i, [\mathcal{S}]_i, [\sigma]_i, [t]_i, [\omega]_i\) for \(i \in \{1, 2\}\) from Lemma 1.

The remaining requirements do not depend on the typing context.

If IMPl\(^2\) command \(c\) is well-typed, then both projections of \(c\) are well-typed (in the IMPl type system).
Lemma 12. If  
\[ pc, \Gamma; C, \Delta, E, H \vdash_{c} c \triangleright \Gamma'; C', \Delta', E', H' \]
then for \( i \in \{1, 2\} \) we have  
\[ pc, [\Gamma]; [C]_{i}, [\Delta]_{i}, [E]_{i}, H \vdash [c]_{i} \triangleright \Gamma'; C', \Delta', E', H' \]

Proof. By induction on the judgment \( pc, \Gamma; C, \Delta, E, H \vdash_{c} c \triangleright \Gamma'; C', \Delta', E', H' \), and the fact that IMPi2 typing rules are at least as restrictive as their IMP counterparts. \( \square \)

The execution of a IMPi2 program preserves typings. The following type-preservation theorem is key to showing that well-typed IMPi programs satisfy delimited release.

**Theorem 2** (Type preservation). Let \( c \) be an IMPi2 command such that  
\[ pc, \Gamma; C, \Delta, E, H \vdash_{c} c \triangleright \Gamma'; C', \Delta', E', H' \]
and let \( (c, \sigma, t, \omega) \) be an IMPi2 configuration such that  
\[ \Gamma, C, \Delta, E \vdash_{c} (c, \sigma, t, \omega) \]
If  
\[ (c, \sigma, t, \omega) \rightarrow (c', \sigma', t', \omega') \]

**Figure 6.** Typing rules
and \([t']_1\) and \([t']_2\) agree up to \(\epsilon\) on escape hatches \(H'\) then there exists \(C'', \Delta'', E'', H'', \Gamma'', \) and \(pc''\) such that

\[pc'', \Gamma''; C'', \Delta'', E'', H'' \parallel_\epsilon c' \triangleright \Gamma', C', \Delta', E', H'\]

and

\[\Gamma'', C'', \Delta'', E'' \parallel_\epsilon (c', \sigma', t', \omega)\).

**Proof.** By induction on the typing judgment \(pc, \Gamma; C, \Delta, E, H \parallel_\epsilon c \triangleright \Gamma'; C', \Delta', E', H'\).

First, note that

\[\{C'[i], [\Delta'[i]], [E'[i]], ([\sigma'[i], [\ell'[i]], [\omega'[i]]])\}

for \(i \in \{1, 2\}\) by Lemmas 12 and 2 (since the witnesses chosen in the proof of Lemma 2 are the same as the witnesses chosen in this proof).

- **T2-SKIP**

  Trivial, as there is no \((c', \sigma', t', \omega)\) such that \((\text{skip}, \sigma, t, \omega) \rightarrow (c', \sigma', t', \omega)\).

- **T2-ASSIGN**

  In this case, \(c = x \vdash c', \sigma' = \sigma(x \rightarrow \sigma(e)), t' = t, C' = C, \Delta' = \Delta[x \rightarrow \Delta(e)], E' = E, H' = H, \Gamma' = \Gamma[x \rightarrow \ell]\).

  Let \(C'' = C', \Delta'' = \Delta', E'' = E', H'' = H', \Gamma'' = \Gamma'\).

  Then \(pc, \Gamma''; C'', \Delta'', E'', H'' \parallel_\epsilon c' \triangleright \Gamma'; C', \Delta', E', H'\) as required.

  Since \(\Gamma, \Delta, E \vdash (c, \sigma, t, \omega)\) we have for all \(y \in \text{Var}\) such that \(y \neq x\), if \(\sigma(y)\) is an integer pair, then \(\Gamma'(y) = \Gamma(y) \nsubseteq \ell\). Also, if \(\sigma(x)\) is an integer pair, then \(\sigma(e)\) is an integer pair, and so there is some variable \(z \in \text{Var}\) such that \(\Gamma(z) \nsubseteq \ell\). By T2-ASSIGN, we have \(\Gamma(z) \nsubseteq \Gamma'(z)\), and so \(\Gamma'(z) \nsubseteq \ell\).

- **T2-SEQ**

  In this case, \(c = c_0; c_1\). If \(c_0 \neq \text{skip}\) then the result follows trivially from the inductive hypothesis. If \(c_0 = \text{skip}\) then \(c' = c_1\), and \(\sigma' = \sigma\) and \(t' = t\), and the result follows easily.

- **T2-OUT**

  In this case, \(c = \text{output}\) \(e\) to \(t', c' = \text{skip}, \sigma' = \sigma, t' = t'\text{ output}(t', \sigma(e))\), \(C' = \text{incout}(C, t', \sigma(e)), \Delta' = \Delta, E' = \text{recordoutput}(E, C, t', \Delta(e)), H' = H, H' = H, \Gamma' = \Gamma\).

  Then \(pc, \Gamma''; C'', \Delta'', E'', H'' \parallel_\epsilon c' \triangleright \Gamma'; C', \Delta', E', H'\) as required.

  If \(\sigma'(x)\) is a pair value, then \(v\) is a pair value, and so, by \(\Gamma, C, \Delta, E \vdash (c, \sigma, t, \omega)\), we must have \(\ell' \nsubseteq \ell\). Similarly, if \(\text{value}(\text{output}(t', \sigma))\) is a pair value, then \(\ell' \nsubseteq \ell\). Thus, we have \(\Gamma', C', \Delta', E' \parallel_\epsilon (c', \sigma', t, \omega)\) as required.

- **T2-IF**

  In this case, \(c = \text{if } e \text{ then } c_0 \text{ else } c_1\). Suppose that \(\sigma(e)\) is not a pair value, i.e., \(\sigma(e) \notin \mathbb{Z}\). Then \(c' = c_0, \sigma' = \sigma, t' = t, C' = C, \Delta' = \Delta, E' = E, H' = H, \Gamma' = \Gamma\).

  Let \(C'' = C', \Delta'' = \Delta', E'' = E', H'' = H', \Gamma'' = \Gamma'\).

  Then \(pc, \Gamma''; C'', \Delta'', E'', H'' \parallel_\epsilon c' \triangleright \Gamma'; C', \Delta', E', H'\) and \(\Gamma', C', \Delta', E' \parallel_\epsilon (c', \sigma', t, \omega)\) as required.

  Otherwise, suppose that \(\sigma(e)\) is a pair value. Then \(c' = \{c'_0 | c'_1\}, \sigma' = \sigma, t' = t, C' = C, \Delta' = \Delta, E' = E, H' = H, \Gamma' = \Gamma\).

  Let \(C'' = C', \Delta'' = \Delta', E'' = E', H'' = H', \Gamma'' = \Gamma'\).

  By Lemma 12 and T2-IF, if we have

  \[pc', [\Gamma 1 \vdash [C] 1 \vdash [\Delta] 1 \vdash [E] 1, H \supseteq c_0 \triangleright \Gamma'; C', \Delta', E', H'\]

  for \(i \in \{1, 2\}\) and some \(pc'\) such that \(\Gamma'(e) \subseteq pc'\). Since \(\sigma(e)\) is a pair value, there is some variable \(y \in \text{Var}\) that appears in \(e\) such that \(\sigma(y)\) is a pair value. Then \(c' = \{c'_0 | c'_1\}, \sigma = \sigma, t' = t, C' = C, \Delta' = \Delta, E' = E, H' = H, \Gamma' = \Gamma\).

- **T2-WHILE**

  In this case, \(c = \text{while } e \text{ do } c_0, c'_1 = \text{if } e \text{ then } (c_0 \text{ while } e \text{ do } c'_1)\text{ else } (c'_0, \sigma', t' = t, C' = C, \Delta' = \Delta, E' = E, H' = H, \Gamma' = \Gamma\).

  Let \(C'' = C', \Delta'' = \Delta', E'' = E', H'' = H', \Gamma'' = \Gamma'\).

  We can easily construct a judgment \(pc, \Gamma''; C'', \Delta'', \Delta', E, H' \parallel_\epsilon c' \triangleright \Gamma'; C', \Delta', E', H'\) and moreover \(\Gamma', C', \Delta', E' \parallel_\epsilon (c', \sigma', t, \omega)\) as required.

- **T2-DECLASSIFY**

  In this case, \(c = \text{ x := declasify(e to } t')\), \(c' = \text{ skip, } t' = t, C' = C, \Delta' = \Delta[x \rightarrow \Delta(e)], E' = E, H' = H \cup \{(\Delta(e), t')\}, \Gamma' = \Gamma[x \rightarrow \ell]\).

  If \(c' = (\text{ if e then } \ell)\) then \(\sigma = \sigma[x \rightarrow \ell]\), otherwise \(\sigma = \sigma[x \rightarrow \sigma(e)]\).

  Let \(C'' = C', \Delta'' = \Delta', E'' = E', H'' = H', \Gamma'' = \Gamma'\).

  Then \(pc', \Gamma''; \ell; C'', \Delta'', E'' \parallel_\epsilon c' \triangleright \Gamma'; C', \Delta', E', H'\) as required.

  Note that by noPairs(\(\Gamma, C, \Delta, E\), \(\Delta(e)\) is not a pair of input expressions. By assumption, \([t']_1 \text{ and } [t']_2\) agree up to \(\epsilon\) on escape hatches \(H'\). If \(\ell' \nsubseteq \ell\) then \([t']_1 \parallel \Delta(e) \downarrow v\) and \([t']_2 \parallel \Delta(e) \downarrow v\) for some integer \(v\). Since \(C, \Delta, E, \parallel (c, \sigma, t, \omega)\), we have \(\sigma(e) = v\), and so \(\sigma(x) = v\).
If \( \sigma'(x) \) is a pair value, then \( \sigma(e) = \{ v_1, v_2 \} \) where \( v_1 \neq v_2 \), and so we must have \( \ell' \not\equiv \ell \). Thus, \( \Gamma', C', \Delta', E' \vdash (e', \sigma', t, \omega) \) as required.

- **T\(^2\)-SUB**: By induction on the length of \( t \).
  - By Lemma 11, we have \( \vdash c' \triangleright \Gamma' ; C'_1 ; c'_1 \vdash (e', \sigma, t, \omega) \).
  - By the inductive hypothesis, and Lemma 11, we have \( \vdash c'_1 \triangleright \Gamma' ; C'_1 ; \Delta'_1 ; E'_1 \vdash (e', \sigma, t, \omega) \).

By T\(^2\)-PAIR, we have \( \vdash (e', \sigma, t, \omega) \) as required.

- **T\(^2\)-PAIR**: If \( c = \{ c_1 \} \) then \( c_1 = c_2 \) is skip, then \( \vdash (e', \sigma, t, \omega) \) as required.
  - Otherwise, if \( c \neq \{ c_1 \} \) then \( \vdash (e', \sigma, t, \omega) \) as required.

**Lemma 12**: If \( \vdash (e', \sigma, t, \omega) \) then \( \vdash (e', \sigma, t, \omega) \) as required.

**Proof**. Every IMPI typing rule is made into an IMPI\(^2\) typing rule by adding the typing parameter \( \ell \).

**A2.4 Proof of delimited release**

Using the type preservation of IMPI\(^2\), and the lemmas above, we can now show that a well-typed IMPI program satisfies delimited release.

**Lemma 13**: If \( \vdash (e', \sigma, t, \omega) \) then \( \vdash (e', \sigma, t, \omega) \) as required.

**Proof**. Assume \( \vdash \Gamma, C, \Delta, E, H \vdash (e', \sigma, t, \omega) \) in IMPI\(^2\).

**A2.4.1 Proof of delimited release**

Using the type preservation of IMPI\(^2\), and the lemmas above, we can now show that a well-typed IMPI program satisfies delimited release.

**Lemma 15**: If \( \vdash \Gamma, C, \Delta, E, H \) for some \( \Gamma_0 \) and \( \vdash \Gamma, C, \Delta, E, H \) then \( \vdash \Gamma, C, \Delta, E, H \) for all \( \ell \).
Theorem 2, we have Γ′, C′, ∆′, E′ ⊨ (c′, σ′, t, ω) for some C′, ∆′, E′, and Γ′. Thus, by Lemma 13, we have \(|t_j|_\ell = |t|_\ell\), and so \(t_j|_\ell \succeq t_i|_\ell\) and thus \(t_1 \approx t_2\) as required.

Let one or both of \(t_1\) or \(t_2\) be an infinite trace. Suppose that it is not the case that either \(t_1|_\ell \succeq t_2|_\ell\) or \(t_2|_\ell \succeq t_1|_\ell\). Therefore there is some index \(n\) such that \((t_1|_\ell)(n) \neq (t_2|_\ell)(n)\). Consider finite traces \(t'_1\) and \(t'_2\) such that \(t_1 \succeq t'_1\) and \(t_2 \succeq t'_2\), and \(|t'_1|_\ell| = |t'_2|_\ell| = n + 1\). Note that \((c, \sigma, \langle \rangle, \omega_1) \rightsquigarrow t'_1\) and \((c, \sigma, \langle \rangle, \omega_2) \rightsquigarrow t'_2\). By a similar argument above, we derive that \(t'_1 \approx t'_2\). But this implies that \((t'_1|_\ell)(n) = (t'_2|_\ell)(n)\), a contradiction! Therefore, either \(t_1|_\ell \succeq t_2|_\ell\) or \(t_2|_\ell \succeq t_1|_\ell\), and so \(t_1 \approx t_2\) as required. □

**A.3 Bounded release**

The proof of Theorem 1 follows immediately from Lemmas 7 and 15.