Improving Usability of Information Flow Security in Java

Scott F. Smith   Mark Thober

The Johns Hopkins University
{scott,mthober}@cs.jhu.edu

Abstract

This paper focuses on improving the usability of information flow type systems. We present a static information flow type inference system for Middleweight Java (MJ) which automatically infers information flow labels, thus avoiding the need for a multitude of program annotations. Additionally, policies need only be specified on IO channels, the critical flow boundary. Our type system includes a high degree of parametric polymorphism, necessary to allow classes to be used in multiple security contexts, and to properly distinguish the security policies of different IO channels.

We prove a noninterference property for programs that interactively input and output data. We then describe a mechanism that allows users to define top-level policies, which automatically inserts the security policies at the proper points in the program. This provides the further benefit that whomever is defining the policy does not necessarily need intimate knowledge of the program source.

1 Introduction

While the foundations of static information flow systems are solid, there remains a usability gap that needs to be closed. The overhead for adding information flow security to programs is potentially large, since existing systems usually require that security annotations be added to the code. With large numbers of annotations, the likelihood of having incorrect annotations also increases: a mistake can get lost in the noise of so many annotations. Input/output is another important practical concern which has also not been fully integrated into static information flow systems.

Information flow research [4, 13, 24, 18, 31] has shown how type systems can be defined to statically guarantee that high security data will not affect low security data. A *noninterference* [12] property is usually shown for well-typed programs: low security outputs are not affected by any high security inputs. The majority of these works assume a batch model of IO, although O'Neill *et. al.* recently described a technique for enforcing information flow security for interactive IO, using a simple imperative language and basic type system [21].

Our primary goal is to provide practical data secrecy and integrity protection to aid programmers in securing programs they write. To this end, we present a provably correct static information flow type inference system, for a core subset of Java (namely, Middleweight Java) that automatically infers information flow labels, thus avoiding the need for a multitude of program annotations. Policies need only be specified on IO channels, which we will argue to be the only real flow boundary that must be considered. The type system includes a high degree of parametric polymorphism, necessary to allow classes to be used in multiple security contexts, and to distinguish policies of different IO channels.

Our work places the focus on input and output points as *the* important boundaries for securing data. Thus, we are only indirectly concerned about internal flows, in how they ultimately will relate to the inputs and outputs. In general, we should speak of securing the *component interface* [30], since runtimes may be composed of multiple independent components with distinct security policies; here we focus on just the IO boundary for simplicity.

As is common practice in information flow type systems, we associate a flow label with each program value. Labels are explicitly placed on input data and checking policies explicitly declared at output points; for points in between, the type system automatically infers the labels and so programmers do not need to add declarations. Input statements are of the form $\mathtt{read}_{(\mathtt{L_s},\mathtt{L_i})}(\mathtt{fd})$, where $\mathtt{L_s}$ and $\mathtt{L_i}$ are the declared security level policy for secrecy and integrity of the channel, respectively, and $\mathtt{fd}$ is the file descriptor that names the channel. Similarly, output statements are of the form $\mathtt{write}_{(\mathtt{L_s},\mathtt{L_i})}(\mathtt{e},\mathtt{fd})$. For practicality we also support the ability to downgrade (*declassify*) secrecy labels, and upgrade (*endorse*) integrity labels when deemed safe to do so.

The type inference system provides an expressive form of parametric polymorphism. Polymorphism is crucial for modeling information flows with fine enough granularity. Different objects of the same class (e.g. two completely different

`HashSet` objects) may be used in different security contexts, which must be differentiated in the analysis. Otherwise, secure programs may be rejected by a type system that unnecessarily merges flows. In our system, security policies on IO channels are defined at the level of Java `Stream` classes. This allows a `LowOutputStream` class to have a different security requirement than a `HighOutputStream` class. As described in Section 2, our fine-grained polymorphic type inference algorithm is essential for providing a fine enough distinction on IO channels. To demonstrate the correctness of our system, we prove a type soundness result, and we also show a noninterference property, extended to account for interactive inputs and outputs.

One weakness of Java and other programming languages is how the IO points can get buried in the code through subclassing, method calls, *etc*. This in turn makes it difficult to observe the policies on the use of IO channels without digging through the whole program. This lack of a clear top-level IO interface means anyone who wants to understand the information flow properties of a whole program must have knowledge of the code details in order to understand what information flows occur through IO. We describe a simple mechanism that allows users to define concise top-level policies which are then automatically applied to the proper IO points in the program. This reduces the burden on both the programmer as well as the policy validator – the security policy for the whole program is now defined in one place.

The result of our strong type inference system and IO policy declarations is a usable system for a real language, where programmers need only specify the security policy of IO channels, and the type system ensures the program does not violate the policy.

2 System Overview

Our syntax is based on Middleweight Java (MJ) [8], extended with labeled input and output operations, declassifying and endorse syntax as well as other minor additions. Input and output statements are $\mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\mathtt{fd})$ and $\mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\mathtt{e},\mathtt{fd})$, where `fd` is the file descriptor of the IO channel, `e` is what is written to the output channel, and `L` and $\mathtt{L}'$ are sets of labels specifying the secrecy and integrity levels of the channel, respectively. For convenience, we use labels sets and the usual set relations as our security lattice [11].

At the point of a read operation, the returned value is tagged with the security labels of the channel. Further, checks are performed to ensure it is safe to read in the current security context. For example, a low read must not occur under a high guard. Otherwise, an attacker would notice that the amount of data read from a low stream would differ if the high guard differed. For example, one execution may read from a low stream three times, while another execution with a different high guard may read from the stream seven times, indirectly leaking information in the three vs. seven number. At each write, the labels on the value to be put to the channel are checked against the channel policy, to ensure that high secrecy data is not output to a low secrecy channel (and, dually that low integrity data does not flow into a high integrity channel).

Integrity is an important dimension of information flow security that is often ignored. While most research correctly states that integrity is a dual to secrecy [7], there are subtle differences [15, 17], and for this reason we model both secrecy and integrity in detail. Our goal of providing a usable information flow system provides further motivation for including integrity in our analysis, since integrity is an important dimension of information assurance.

We provide a `Declassify(e,L)` statement, which removes secrecy labels `L` from `e`. This serves to declassify data in infrequent, explicitly allowable instances [20, 34]. For example, in a program where a password is being checked, the result of a password comparison may be declassified, so the resulting boolean will not carry the high security label of the password. Programmers must be very careful when using declassify operations, because they may reveal too much information and compromise security. We also provide the integrity dual, `Endorse(e,L)`, which increases the integrity label of the argument, specifying increased confidence in the data.

We define a static constraint-based type inference system, with a form of automatic label polymorphism inference that is related to CPA-style concrete class analyses [3, 27, 33]. The need for label polymorphism inference will become evident when we study the example program of Section 2.2.

2.1 Program Constants and Default Policies

The use of security-critical constants directly in the program text can create security holes: hard-coded secret data may be mislabeled and leak out of a program through output operations, or by an unauthorized agent reading the source code itself. Similarly, program constants may adversely affect data integrity, *e.g.* if a rogue string constant is inadvertently written as a user's password. Remarkably, programmers continue to make such mistakes, even in recent commercially available programs and devices [25, 2, 10], where hard-coded passwords resulted in security problems.

Example 1 Password Changing Program

```
class PwdFile extends Object {
 String fileName; String tempName;

 bool ChangePwd(String uname,oldpwd,newpwd){
  bool succ = false; String line;
  BR passIn = getPwdReader();
  PrintWriter tempOut = getWriter();
  while((line = passIn.readLine()) != null) {
   if (isUser(line,uname,oldpwd)) {
    tempOut.println(uname + ":" + newpwd);
    succ = true;
   } else { tempOut.println(line) }
  }
  // rename tempFile to fileName
  return Declassify(succ,{high,sys});
 }
 Reader getPwdReader() {
  SysFileIS fin = new SysFileIS(fileName);
  return new BR(new ISReader(fin));
 }
 Writer getWriter() {
  PwdFileOS fout = new PwdFileOS(tempName);
  return new PrintWriter(fout);
 }
 bool isUser(String line, uname, oldpwd) {
  // parse line and return true if uname and oldpwd match
 }
}
```

```
public class SysFileIS extends FileIS {
 public int read() {
  return read_({high,sys},{high,sys})(fd); }
}

public class UserIS extends IS {
 public int read() {
  return Endorse(super.read(),{high});}
}

public class PwdFileOS extends FileOS {
 public void write(int v) {
  write_({high,sys},{high})(v,fd); }
}

void main(){
 String fileName = "/etc/passwd";
 String tempName = "/tmp/tmppasswd";
 PwdFile pf = new PwdFile(fileName,tempName);

 //read uname,oldpwd,newpwd from a UserIS.

 bool succ = pf.ChangePwd(uname,oldpwd,newpwd);
 if (succ) {
  System.out.println("Success");
 } else {
  System.out.println("Failure");
 }
}
```

We take the approach that hard-coding of secret data or low-integrity data simply should not happen: the only reasonable way to view program constants are as low secrecy but high integrity data, and this is how our type system treats all constants.

Establishing default policies for input and output channels is a closely related problem. This is important for establishing security for programs where not all IO channels have been given a security policy, and in describing policies for the standard input and output streams (`System.in`, `System.out` and `System.err` in Java). The default policy for an input channel is established as low secrecy and low integrity. This means the data is considered public and unreliable, which is a natural default for an unknown channel. The default policy for an output channel is also low secrecy and low integrity. This means the channel is considered observable to public users, and does not require any degree of confidence in the integrity of the data being output.

2.2 An Example Java Program

In this section we elaborate on how information flow is controlled at IO points in our system, by the study of a simple example. In the following subsection we then give an overview of our parametric polymorphism and label inference system.

IO channels in Java are created through subclassing, creating classes such as `FileInputStream`, `DataOutputStream`, `SocketInputStream`, etc. We build on this approach by defining different information flow policies via subclassing the core IO classes. In particular, a different subclass is created for each distinct security category of IO. This 1-1 relationship between class definitions and security policies makes for an *object-oriented* approach to information flow policies, harmonizing with the existing language structures.

We now focus on an Example 1, a program for changing passwords, where data security is important in both secrecy and integrity dimensions. This example is somewhat oversimplified but is short enough to illustrate the key concepts. Firstly, we want to provide secrecy for the user name and password information contained on the system, making sure this information

Example 2 Password Changing with Polymorphism

```
...
 bool ChangePwd(IS in,OS out,String uname,
  String oldpwd, String newpwd){
  bool succ = false; String line;
  BR passIn = new BR(new ISReader(in));
  PrintWriter tempOut = new PrintWriter(out);
  // ... same code as above
 }
...
public class TopFileOS extends FileOS {
 public void write(int v) {
  write({top,high,sys},{top,high})(v,fd);}}
```

```
void main() {
 // ... same code as above
 String ts = "/etc/topsecret";
 SysFileIS in = new SysFileIS();
 PwdFileOS pout = new PwdFileOS(tempName);
 TopFileOS tout = new TopFileOS(ts);
 pf.ChangePwd(in,pout,uname,oldpwd,newpwd);
 pf.ChangePwd(in,tout,uname,oldpwd,newpwd);
}
```

is not leaked to a public channel, *i.e.* the screen. Secondly, we want to ensure the integrity of the system password file by not allowing it to be tainted by improper data, thereby altering user names and passwords on the system. These are two well-defined goals for a programmer of a password changing application.

We take some liberties with syntax that is not described in our calculus, such as the use of local variables, `super()`, and a `while` loop. We make some abbreviations to shorten the presentation, `IS` for `InputStream`, `OS` for `OutputStream`, `PS` for `PrintStream`. `B` abbreviates `Buffer`, and `BR` is `BufferedReader`. Other obvious abbreviations have been made, and some code is omitted.

The modifications needed to support information flow analysis here are minor. The most significant requirement is to define distinct subclasses of `InputStream` and `OutputStream` for each distinct IO policy. In this case we are defining three new IO policies, in the classes `SysFileIS` and `UserIS` (for input), and `PwdFileOS` (for output). For `SysFileIS`, the `read` method labels input values with `high` and `sys` for both secrecy and integrity. The `write` method of `PwdFileOS` allows secrecy labels `high` and `sys`, and requires the integrity label `high`, thereby enforcing the policy that only certain data may be written to the password file. The `UserIS` class is defined with an `Endorse` operation, expressing confidence in the integrity of the data on the channel. (Note that IO can occur with other methods such as file `rename`, but we are simplifying a bit in this example). There is also a declassification of secrecy labels at the end of the `ChangePwd` method, necessary to allow the success or failure of the program to be output to the screen.

Note that this program shows how code is written in the language, no explicit parametric type declarations are needed, and no label type declarations need to be placed on variables – type parametricity and variable information flow labels are both inferred automatically. So, the underlying Java program only needs to be changed to declare the appropriate IO channels and policies, and to add any needed downgrading and upgrading constructs. The underlying program structure remains largely unchanged, *e.g.* a `SysFileIS` object `sysin` is still accessed via `sysin.read()`, with no need for annotation.

Proper typing of this example imposes some requirements on the type system: the type of the `read` and `write` methods simply *cannot* be the same across all subclasses, otherwise all of the work we made to separate the policies in separate classes would be for nothing since the type system would merge the information flows. So, a form of parametric polymorphism is needed to distinguish between subclasses. It is even more subtle because a variable declared to be an `InputStream` can at runtime be any of its subclasses such as `SysFileIS` or `UserIS`, and so it may look very difficult to type these methods distinctly. Our solution is to use a polymorphic form of concrete class analysis [3]: we use a constraint-based type system that specializes the type of an object at each method call site for each different type of object that it could be. This technique leads to a very accurate typing [3, 33], and allows the methodology of placing different security policies in different subclasses to be sound yet expressive. The most obvious forms of polymorphic type inference, based on treating each class or interface as polymorphic and not each method and message send, are too weak to properly treat examples such as the `InputStream` mentioned above.

2.3 Polymorphism

To better illustrate the expressiveness of our polymorphic type system we show an alternate implementation of the `ChangePwd` method in Example 2. This implementation takes an `InputStream` and `OutputStream` as arguments for reading from and

Example 3 HashSet Polymorphism

```
public class HighFileIS extends FileIS {
 public int read() {
  return read$_{(\{\mathrm{high}\},\{\mathrm{high}\})}$(fd); }}

public class LowFileIS extends FileIS {
 public int read() {
  return read$_{(\emptyset,\emptyset)}$(fd); }}

public class LowFileOS extends FileOS {
 public void write(int v) {
  write$_{(\emptyset,\emptyset)}$(v,fd); }}
```

```
void main() {
 HashSet highSet = new HashSet();
 FileIS hin = new HighFileIS("high_infile");
 int i;
 while(i = hin.read()) {
  highSet.add(i);
 }
 HashSet lowSet = new HashSet();
 FileIS lin = new LowFileIS("low_infile");
 int j;
 while(j = lin.read()) {
  lowSet.add(i);
 }
 Iterator lowIt = lowSet.iterator();
 FileOS lowout = new LowFileOS("low_outfile");
 lowout.write(lowIt.next());
}
```

writing to the password file, respectively. The `main` method in Example 2 uses this new implementation. `TopFileOS` is subclassed from `FileOS`, and the `write` method of the new class checks the output data for the integrity label `Top`. In the `main` portion, two different calls are made to `ChangePwd`, one with a `PwdFileOS`, as before, and one to a `TopFileOS`.

Our polymorphic type system is expressive enough to directly support this new `ChangePwd` method. Additionally, since we are statically inferring the concrete classes of objects, we can create different security policies for overriding methods, and the type system will know the correct policy to use. In this example, the first call to `ChangePwd` will type properly, but the second call will cause a type error, since the data passed to the `write` method of the `TopFileOS` is not labeled with `top`.

In addition to the need for polymorphism for discriminating input and output streams, we also need polymorphism for code re-use. Code should be reusable in multiple contexts, and those contexts may also have different information flow policies. This means concretely that library classes and methods must be allowed to be instantiated at multiple security contexts, and the type system must not merge all of the flows. We illustrate this with the program in Example 3, which uses different `HashSet` objects: one holding high data, and the other holding low data.

We define two input stream classes, one for reading in high data, and one for low data, and an output stream class for writing low data. The program reads data from both high and low streams and puts them in separate `HashSet` objects. A value is then taken from the `HashSet` containing low data, and written to the low output channel.

This clearly shows the need for polymorphism over security levels. If the types for these two `HashSet` objects were merged, the program would be rejected, because high data would appear to flow out a low channel. Our system views `HashSet` as polymorphic and the `highSet` and `lowSet` are typed distinctly, so the program typechecks.

3 Types for Data Tracking and Checking

We now present the formal type inference system. In order to simplify the reasoning and presentation of the system, we define a *label type inference system* solely for typing data flows, and use the existing MJ type system for normal MJ typechecking not related to information flow. Our label type system is strong enough to handle any valid MJ program, including those with mutually recursive class definitions, and method recursion. A program type checks if and only if it type checks in both the MJ type system and the label type system.

3.1 The Language

Our language is an extension of Middleweight Java (MJ) [8]. MJ contains the basic object constructs of Java, including state; it omits some of the more complex features of Java, which allows formal properties to be established. We eliminate local variables, which complexify the operational semantics and proofs, although their typings are a straight forward extension of

P	::=	$\bar{\mathtt{CL}}; \bar{\mathtt{s}}$	*program*
CL	::=	class C extends C $\{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\}$	*class*
K	::=	$\mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{super}(\bar{\mathtt{e}}); \bar{\mathtt{s}}\}$	*constructor*
M	::=	$\mathtt{RT}\ \mathtt{m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\bar{\mathtt{s}}\}$	*method*
RT	::=	C $\mid$ void	*return type*
L	::=	$\{\bar{\mathtt{l}}\}$, where l are unique labels.	*label*
CO	::=	c $\mid$ b $\mid$ str $\mid$ null $\mid$ fd	*constant*
e	::=	x $\mid$ this $\mid$ CO $\mid$ e.f $\mid$ (C) e $\mid$	*expression*
		e $\oplus$ e $\mid$ pe $\mid$ Declassify(e, L) $\mid$	
		Endorse(e, L) $\mid$ $\mathtt{read}_{(\mathtt{L},\mathtt{L})}(\mathtt{fd})$	
pe	::=	$\mathtt{e.m}(\bar{\mathtt{e}})$ $\mid$ new $\mathtt{C}(\bar{\mathtt{e}})$ $\mid$	*promotable exp.*
s	::=	pe; $\mid$ if e then $\{\bar{\mathtt{s}}\}$ else $\{\bar{\mathtt{s}}\}$ $\mid$	*statement*
		; $\mid$ $\{\bar{\mathtt{s}}\}$ $\mid$ e.f := e; $\mid$ return e; $\mid$	
		$\mathtt{write}_{(\mathtt{L},\mathtt{L})}(\mathtt{e}, \mathtt{fd})$	

Figure 1: Grammar

object fields. We add *constants* (int, bool, string, file descriptor), *operators* (+,- etc.), in order to better reason about information flows in real programs. We also add low level read and write operations to the language, of the form $\mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\mathtt{fd})$ and $\mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\mathtt{e},\mathtt{fd})$, where fd is the file descriptor of the IO channel, e is what is written to the output channel, and L and L′ are sets of labels specifying the secrecy and integrity levels of the channel, respectively. We also add a Declassify(e,L) construct, which removes the secrecy labels in L from those on e. Endorse(e,L) is the integrity dual of declassification that adds integrity labels L to those on e. The grammar for our Extended MJ (EMJ) language is given in Figure 1.

We assume some familiarity with MJ, and do not reproduce its typing or semantic definitions; see [8] for the details. Note that EMJ follows MJ and types expressions with respect to a global class table, CT, that contains the types of all classes. At the top level a sequence of statement $\bar{\mathtt{s}}$ corresponding to the main method is typechecked with respect to this table.

In MJ, type assertions are of the form $CT; \Gamma \vdash_T \mathtt{e} : \mathtt{C}$ meaning under class table CT and type environment Γ, expression e has type C (note this Γ is different from our definition). A similar definition is given for statements. In addition to the standard type rules for MJ, we add the type rules corresponding to the EMJ extensions; they are mostly straightforward, and are omitted for lack of space. $\mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\mathtt{fd})$ is typed to input an integer and $\mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\mathtt{e},\mathtt{fd})$ outputs an integer (e has an integer type), while fd is of type FileDescriptor. For Declassify(e, L) and Endorse(e, L), the resulting type of the expression is the same type as e, since the label tracking is only handled in the label typing rules.

3.2 Label Types

EMJ values are either objects or primitive constants. Objects may be labeled, as may the internal fields of an object. Thus, Label types, τ, are four-tuples $\langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$; $\mathcal{S}$ is a set of secrecy labels for the current object, $\mathcal{I}$ is a set of integrity labels for the object, $\mathcal{F}$ is a record containing sets of labels, representing the internal fields of the object, and $\mathcal{A}$ is an α-type, a type representing the concrete class of the object, explained below. The type definitions are summarized in Figure 2.

An object's fields has its own labels, represented by the field type $\mathcal{F}$, which is a mapping of field names to types, $\{\mathtt{f}_1 \mapsto \tau_1, \ldots, \mathtt{f}_n \mapsto \tau_n\}$. The individual labels may be accessed by a dot notation: $\mathcal{F}.\mathtt{f}.\mathsf{S}$ is the secrecy label on the f field of the object. Primitive constants are labeled as objects with no fields.

The α-types are used to express a form of parametric polymorphism over the inheritance hierarchy, allowing the superclass and subclass to differ in their labeling. The usual Java type declaration is insufficient for determining the class of an object, as it may be an object of a subclass, which contains a different policy, or returns different labels. As discussed in Section 2, we need a more expressive form of polymorphism. We employ an analysis that is closely related to Data-polymorphic CPA [27, 33], a variant of CPA [3]. This ensures proper creation of distinct *contours* (polyinstantiations) when needed to give the type expressivity required for our system, while on the other hand merging enough contours to make sure the analysis terminates.

We use l to represent a concrete label, and s, i for label variables in the secrecy and integrity domains, respectively. Notation L refers to a set of concrete labels $\{\bar{\mathtt{l}}\}$, and label sets $\mathcal{S}, \mathcal{I}$ may contain both concrete label sets and label variables, the latter used when the concrete label is not yet known. For example, when typing methods, the argument labels are variables

τ	$::=$	$\langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \mid t$	*types*
$\mathcal{S}$	$::=$	$\{\bar{\mathtt{l}}\} \mid s^\sigma \mid \mathcal{S} \cup \mathcal{S} \mid \mathcal{S} - \mathcal{S} \mid \mathcal{F}.\mathtt{f}.\mathsf{S} \mid \emptyset$	*secrecy types*
$\mathcal{I}$	$::=$	$\{\bar{\mathtt{l}}\} \mid i^\sigma \mid \mathcal{I} \cap \mathcal{I} \mid \mathcal{I} \cup \mathcal{I} \mid \mathcal{F}.\mathtt{f}.\mathsf{I} \mid \emptyset$	*integrity types*
$\mathcal{F}$	$::=$	$\{\bar{\mathtt{f}} \mapsto \bar{\tau}\} \mid f^\sigma \mid \mathcal{F}.\mathtt{f}.\mathsf{F} \mid \emptyset$	*field types*
$\mathcal{A}$	$::=$	$\mathtt{C} \mid \alpha^\sigma \mid \mathcal{F}.\mathtt{f}.\mathsf{A}$	*alpha types*
σ	$::=$	$\mathtt{C}, \mathtt{m}, \bar{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r \mid \epsilon$	*contours*
$s^\sigma, i^\sigma, f^\sigma, \alpha^\sigma, s_p, i_p$			*label variables*
t	$::=$	$\langle s^\sigma, i^\sigma, f^\sigma, \alpha^\sigma \rangle$	*type variables*
κ	$::=$	$\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C}$	*method types*
		$\forall \bar{t}'.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C}$	
c	$::=$	$\mathcal{S} <: \mathcal{S} \mid \mathcal{I} <: \mathcal{I} \mid \mathcal{F} <: \mathcal{F}$	*constraints*
		$\mid \mathcal{A} <: \mathcal{A} \mid SC(\mathtt{L}, \mathcal{S}) \mid IC(\mathtt{L}, \mathcal{I})$	
		$\mid \mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc, pc_i} \tau_r)$	
		$\mid \tau <: \textbf{get}\ \tau \mid \tau <: \textbf{set}\ \tau$	
$\mathcal{C}$	$::=$	$\{c\} \mid \mathcal{C} \cup \mathcal{C} \mid \emptyset$	*constraint sets*
pc	$::=$	$\mathcal{S}$	*prog. counter*
pc_i	$::=$	$\mathcal{I}$	*prog. counter*
PC	$::=$	pc, pc_i	*pc abbrev.*
u			*top integrity label*
s	$::=$	$s^\sigma \mid \mathsf{f}.\mathtt{f}.\mathsf{S}$	*trans. types*
i	$::=$	$i^\sigma \mid \mathsf{f}.\mathtt{f}.\mathsf{I}$	*trans. types*
f	$::=$	$f^\sigma \mid \mathsf{f}.\overline{\mathtt{f}.\mathsf{F}}$	*trans. types*
a	$::=$	$\alpha^\sigma \mid \mathsf{f}.\mathtt{f}.\mathsf{A}$	*trans. types*
$\tau <: \tau'$ is short for $\mathcal{S} <: \mathcal{S}', \mathcal{I} <: \mathcal{I}', \mathcal{F} <: \mathcal{F}', \mathcal{A} <: \mathcal{A}'$			

Figure 2: Type Definitions

since the actual labels are not instantiated until the method is invoked. Additionally, f is a field variable referring to abstract fields of an object, and $\mathcal{F}$ is either an abstract or a concrete field mapping; α is a variable referring to an unknown class, and $\mathcal{A}$ is either an abstract class α or a concrete class $\mathtt{C}$. σ defines the contours necessary for polymorphic method typing, and type variables are extended to allow a contour superscript, (e.g. s^σ) and ϵ represents no superscript. For convenience, we generally omit the superscript on variables when it is unimportant. t denotes a full four-tuple of label types, and is simply short-hand.

We implicitly work over a simple equational theory of sets in typing and constraint closure. Concrete unions, $\mathcal{S} \cup \mathcal{S}'$, where $\mathcal{S} = \{\bar{\mathtt{l}}\}$ and $\mathcal{S}' = \{\bar{\mathtt{l}}'\}$ are considered equivalent to the unioned set, $\mathcal{S} \cup \mathcal{S}' = \{\bar{\mathtt{l}}, \bar{\mathtt{l}}'\}$ (without repeats). An analogous equivalence holds for $\mathcal{I} \cap \mathcal{I}'$ when $\mathcal{I}$ and $\mathcal{I}'$ are concrete label sets. $\mathcal{S} - \mathcal{S}'$ is also equivalent to the obvious set difference when both are concrete label sets. For field access, $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.\mathsf{S}$ is equivalent to $\mathcal{S}_i$, where $\mathtt{f}_i \mapsto \langle \mathcal{S}_i, \mathcal{I}_i, \mathcal{F}_i, \mathcal{A}_i \rangle$. A similar equivalence analogously holds for any $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.\mathsf{I}$, $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.\mathsf{F}$, or $\{\bar{\mathtt{f}} \mapsto \bar{\tau}\}.\mathtt{f}_i.\mathsf{A}$.

We use a label table, LT, to keep track of the label types of all classes when typing expressions. This is analogous to the class table CT of the MJ type system that keeps track of all class types. However, since we are inferring label types here, we must build up the label table while typing the classes, as discussed in Section 3.2.4.

Label type rules are of the form $\Gamma, PC \vdash \mathtt{e} : \tau \backslash \mathcal{C}$ and $\Gamma, PC \vdash \mathtt{s} : \tau \backslash \mathcal{C}$, meaning in label environment Γ, with program counters pc and pc_i (PC is short-hand for pc, pc_i), expression $\mathtt{e}$ (or statement $\mathtt{s}$) has label type τ with constraint set $\mathcal{C}$. Γ binds variables to label type variables, $\Gamma(x) = t$. Separate program counters for secrecy and integrity are pc and pc_i, respectively. They track implicit flows through programs and are a standard feature of information flow type systems.

The constraint set, $\mathcal{C}$, contains normal subtyping constraints $<:$ for secrecy, integrity, field, and α-types. In addition, check constraints of the form $SC(\mathtt{L}, \mathcal{S})$, and $IC(\mathtt{L}, \mathcal{I})$, for secrecy and integrity checks, respectively, are placed in $\mathcal{C}$ and the closure process will need to verify their correctness. Method constraints $\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc, pc_i} \tau_r)$ contain the necessary information to tie up method invocations with the labels of the resulting method call. Methods in the label table are universally quantified, $\forall \bar{t}'.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C}$, so they may vary parametrically. This allows distinct contours to be formed for each combination of argument type and call site. We detail this analysis when discussing the constraint closure in section 3.2.5.

We proceed by discussing specific elements of the type inference system separately.

$$\frac{\Gamma(\mathtt{x}) = \langle\, s, i, f, \alpha \,\rangle}{\Gamma, PC \vdash \mathtt{x} : \langle\, s \cup pc, i \cap pc_i, f, \alpha \,\rangle\backslash\emptyset}\text{(Var)} \qquad \frac{\Gamma(\mathtt{this}) = \langle\, s, i, f, \alpha \,\rangle}{\Gamma, PC \vdash \mathtt{this} : \langle\, s \cup pc, i \cap pc_i, f, \alpha \,\rangle\backslash\emptyset}\text{(This)}$$

$$\frac{}{\Gamma, PC \vdash \mathtt{c} : \langle\, pc, pc_i, \emptyset, \mathtt{int} \,\rangle\backslash\emptyset}\text{(Const)}$$

$$\frac{\Gamma, PC \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C} \qquad t \text{ consist of fresh type variables.}}{\Gamma, PC \vdash \mathtt{e.f} : t\backslash\langle\, \mathcal{S} \cup \mathcal{F}.\mathtt{f}.\mathsf{S}, \mathcal{I} \cap \mathcal{F}.\mathtt{f}.\mathsf{I}, \mathcal{F}.\mathtt{f}.\mathsf{F}, \mathcal{F}.\mathtt{f}.\mathsf{A} <: \textbf{get}\ t \,\rangle \cup \mathcal{C}}\text{(Field)}$$

$$\frac{\Gamma, PC \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C}}{\Gamma, PC \vdash (\mathtt{C})\ \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C}}\text{(Cast)} \qquad \frac{\begin{array}{c}\Gamma, PC \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \emptyset, \mathcal{A} \,\rangle\backslash\mathcal{C} \\ \Gamma, PC \vdash \mathtt{e}' : \langle\, \mathcal{S}', \mathcal{I}', \emptyset, \mathcal{A}' \,\rangle\backslash\mathcal{C}'\end{array}}{\Gamma, PC \vdash \mathtt{e} \oplus \mathtt{e}' : \langle\, \mathcal{S} \cup \mathcal{S}', \mathcal{I} \cap \mathcal{I}', \emptyset, \mathtt{int} \,\rangle\backslash\mathcal{C} \cup \mathcal{C}'}\text{(Op)}$$

$$\frac{\begin{array}{c}\Gamma, PC \vdash \mathtt{e} : \tau = \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C} \qquad \Gamma, PC \vdash \bar{\mathtt{e}} : \bar{\tau}\backslash\bar{\mathcal{C}} \\ t_r = \langle\, s, i, f, \alpha \,\rangle \qquad s, i, f, \alpha \text{ are fresh variables}\end{array}}{\Gamma, PC \vdash \mathtt{e.m}(\bar{\mathtt{e}}) : \langle\, \mathcal{S} \cup s, \mathcal{I} \cap i, f, \alpha \,\rangle\backslash\mathcal{C} \cup \bar{\mathcal{C}} \cup \{\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau \xrightarrow{pc\cup\mathcal{S}, pc_i\cap\mathcal{I}} t_r)\}}\text{(Invoke)}$$

$$\frac{\Gamma, PC \vdash \bar{\mathtt{e}} : \bar{\tau}\backslash\bar{\mathcal{C}} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}} \qquad \bar{t}, t_r \text{ consist of fresh type variables}}{\Gamma, PC \vdash \mathtt{new\ C}(\bar{\mathtt{e}}) : \langle\, pc, pc_i, \{\bar{\mathtt{f}} : \bar{t}\}, \mathtt{C} \,\rangle\backslash\bar{\mathcal{C}} \cup \{\mathtt{C.K}(\bar{\tau}, \langle\, pc, pc_i, \{\bar{\mathtt{f}} : \bar{t}\}, \mathtt{C} \,\rangle \xrightarrow{pc, pc_i} t_r)\}}\text{(New)}$$

$$\frac{\Gamma, PC \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C}}{\Gamma, PC \vdash \mathtt{Declassify(e,L)} : \langle\, pc \cup (\mathcal{S} - \mathtt{L}), \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C}}\text{(Declassify)}$$

$$\frac{\Gamma, PC \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C}}{\Gamma, PC \vdash \mathtt{Endorse(e,L)} : \langle\, \mathcal{S}, pc_i \cap (\mathcal{I} \cup \mathtt{L}), \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C}}\text{(Endorse)}$$

$$\frac{\Gamma, PC \vdash \mathtt{e} : \langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \,\rangle\backslash\mathcal{C}}{\Gamma, PC \vdash \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\mathtt{e}) : \langle\, \mathcal{S} \cup \mathtt{L}, \mathcal{I} \cap \mathtt{L}', \emptyset, \mathtt{int} \,\rangle\backslash SC(\mathtt{L}, \mathcal{S}) \cup IC(\mathtt{L}', \mathcal{I})}\text{(Input)}$$

Figure 3: Label Type Rules for Expressions

3.2.1 Expression Typing

The label type inference rules for expressions are given in Figure 3. Here are a few highlights of the rules. (Const) types constants as label types containing only pc for secrecy, and pc_i for integrity, reflecting our view that constants should by default have no secrecy and full integrity as discussed in section 2.1.

In (Field), we use a **get** constraint to obtain the type of a field access. These constraints are discussed further in section 3.2.3. The secrecy and integrity types include the labels on the field within the object, along with the labels the object itself carries.

In (Invoke), the constraint $\mathcal{A}.\mathtt{m}(\bar{\tau}, \tau \xrightarrow{pc\cup\mathcal{S}, pc_i\cap\mathcal{I}} t_r)$ is added to the constraint set. $\mathcal{S}$ and $\mathcal{I}$ are added to the program counters, since the execution of method `m` depends on the object to which the method is being passed. The method type eventually needs to be looked up in the global label table LT. However, since $\mathcal{A}$ may at this point be of unknown class we postpone this decision until more information is known about $\mathcal{A}$, at constraint closure. The above type constraint records the method call information so it can be propagated in the closure once the concrete class of $\mathcal{A}$ is known.

In (New), the names of the fields in the class `C` are looked up using *fields*. We cannot simply add the types of each argument to the field types, since the constructor may not have this behavior. Thus, fresh type variables are created for each field, and the $\mathcal{F}$ element of the type contains these variables. A constraint is added to capture the call to the constructor, which is similar to a method call. The α-type is given the concrete class name of the object being created. pc and pc_i are the secrecy and integrity labels on the new object, respectively. Like constants, objects are assumed to have no secrecy and full integrity by default.

As expected, `Declassify(e,L)` removes `L` from the secrecy labels of `e` in (Declassify), while `Endorse(e,L)` adds `L` to the integrity labels of `e` in (Endorse).

The type of a $\mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\mathtt{e})$ expression contains the security levels of the statement combined with the labels on the file

$$\frac{}{\Gamma, PC \vdash \texttt{;} : \langle pc, pc_i, \emptyset, \texttt{void} \rangle \backslash \emptyset}\text{(No-op)} \qquad \frac{\Gamma, PC \vdash \texttt{e} : \tau \backslash \mathcal{C}}{\Gamma, PC \vdash \texttt{e;} : \tau \backslash \mathcal{C}}\text{(PE)}$$

$$\frac{\Gamma, pc, pc_i \vdash \texttt{e} : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C} \qquad \Gamma, pc \cup \mathcal{S}, pc_i \cap \mathcal{I} \vdash \bar{\texttt{s}_1} : \langle \mathcal{S}_1, \mathcal{I}_1, \mathcal{F}_1, \mathcal{A}_1 \rangle \backslash \mathcal{C}' \quad \Gamma, pc \cup \mathcal{S}, pc_i \cap \mathcal{I} \vdash \bar{\texttt{s}_2} : \langle \mathcal{S}_2, \mathcal{I}_2, \mathcal{F}_2, \mathcal{A}_2 \rangle \backslash \mathcal{C}''}{\Gamma, pc, pc_i \vdash \texttt{if e then } \{\bar{\texttt{s}_1}\} \texttt{ else } \{\bar{\texttt{s}_2}\} : \langle \mathcal{S} \cup \mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{I} \cap \mathcal{I}_1 \cap \mathcal{I}_2, \emptyset, \texttt{void} \rangle \backslash \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}''}\text{(If)}$$

$$\frac{\Gamma, PC \vdash \texttt{e} : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C} \qquad \Gamma, PC \vdash \texttt{e}' : \langle \mathcal{S}', \mathcal{I}', \mathcal{F}', \mathcal{A}' \rangle \backslash \mathcal{C}'}{\Gamma, PC \vdash \texttt{e.f} := \texttt{e}'; : \langle \mathcal{S} \cup \mathcal{S}', \mathcal{I} \cap \mathcal{I}', \emptyset, \texttt{void} \rangle \backslash \mathcal{C} \cup \mathcal{C}' \cup \{\mathcal{F}.\texttt{f} <: \textbf{set}\ \langle \mathcal{S} \cup \mathcal{S}', \mathcal{I} \cap \mathcal{I}', \mathcal{F}', \mathcal{A}' \rangle\}}\text{(F-Assign)}$$

$$\frac{\Gamma, PC \vdash \texttt{s} : \tau \backslash \mathcal{C} \qquad \Gamma, PC \vdash \bar{\texttt{s}} : \tau' \backslash \mathcal{C}' \qquad \texttt{s} \neq \texttt{C x}}{\Gamma, PC \vdash \texttt{s}; \bar{\texttt{s}} : \tau' \backslash \mathcal{C} \cup \mathcal{C}'}\text{(Seq)} \qquad \frac{\Gamma, PC \vdash \bar{\texttt{s}} : \tau \backslash \mathcal{C}}{\Gamma, PC \vdash \{\bar{\texttt{s}}\} : \tau \backslash \mathcal{C}}\text{(Block)}$$

$$\frac{\Gamma, PC \vdash \texttt{e} : \tau \backslash \mathcal{C}}{\Gamma, PC \vdash \texttt{return e;} : \tau \backslash \mathcal{C}}\text{(Return)}$$

$$\frac{\Gamma, PC \vdash \texttt{e} : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C} \qquad \Gamma, PC \vdash \texttt{e}' : \langle \mathcal{S}', \mathcal{I}', \mathcal{F}', \mathcal{A}' \rangle \backslash \mathcal{C}'}{\Gamma, PC \vdash \texttt{write}_{(\texttt{L},\texttt{L}')}(\texttt{e}, \texttt{e}') : \langle \mathcal{S} \cup \mathcal{S}', \mathcal{I} \cap \mathcal{I}', \emptyset, \texttt{void} \rangle \backslash \mathcal{C} \cup \mathcal{C}' \cup SC(\texttt{L}, \mathcal{S} \cup \mathcal{S}') \cup IC(\texttt{L}', \mathcal{I} \cap \mathcal{I}')}\text{(Output)}$$

Figure 4: Label Type Rules for Statements

descriptor argument. Secrecy and integrity checking constraints are also added to the constraint set. There are two reasons for this. Firstly, the constraints ensure that low reads are not happening under high guards; as discussed previously, this may cause an information leak (note the type of any sub-expression implicitly contains the types of the program counters, a fact easily shown by structural induction on e, observing the base cases all add pc, pc_i to the types). Secondly, if the file descriptor value has a higher label than the channel policy, performing the read may result in a security leak (*e.g.*, two executions that differ only in high inputs may read from different low channels, since the file descriptor for the channel differs).

3.2.2 Statement Typing

The type rules for statements are given in Figure 4. In rule (If), the secrecy and integrity types of the condition are added to the respective program counters when typing each branch. (F-Assign) adds a **set** constraint to the constraint set to set the flow of labels into an object field. These constraints are described in section 3.2.3. Typing a $\texttt{write}_{(\texttt{L},\texttt{L}')}(\texttt{e}, \texttt{e}')$ statement produces secrecy and integrity check constraints to ensure the type of the output aligns with the policy of the channel. The type of the file descriptor is also checked against the policy for the same reasons as `read`, discussed earlier. The remaining rules are straightforward.

Statements that have a `void` α-type (e.g. Output, If) could also have empty secrecy and integrity types. They are an artifact of our proof technique. Inclusion of these labels does not affect the typability of programs, since they only occur on statements; since statements cannot be passed as arguments to reads or writes, these labels will never affect a check constraint.

3.2.3 Get and Set Constraints

We use **get** constraints when typing fields in (Field), and **set** constraints for field assignment in (F-Assign). Constraint closure rules (Get) and (Set) ensures that values assigned to a field flow to any read-point of the field, while ensuring that no backward-flows occur in the types [27, 33]. For example,

$$\begin{aligned}&\texttt{x} := \texttt{read}_{(\{\texttt{low}\},\{\texttt{low}\})}(\texttt{fd});\\ &\texttt{z} := \texttt{x};\\ &\texttt{z} := \texttt{read}_{(\{\texttt{high}\},\{\texttt{high}\})}(\texttt{fd}');\end{aligned}$$

will not result in x having the secrecy type $\{\texttt{high}\}$.

Initial Label Table:

$$\frac{\begin{array}{c}\bar{t}, t_t, t_r, s_p, i_p \text{ consist of fresh variables.}\\ \text{Each use of } \mathit{InitialMethod}() \text{ creates distinct variables.}\end{array}}{\mathit{InitialMethod}() = \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \emptyset} \qquad \frac{\begin{array}{c}\bar{t}, t_t, t_r, s_p, i_p \text{ consist of fresh variables.}\\ \text{Each use of } \mathit{InitialConstructor}() \text{ creates distinct variables.}\end{array}}{\mathit{InitialConstructor}() = \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \emptyset}$$

$$\frac{\kappa_i = \mathit{InitialConstructor}() \qquad \bar{\kappa}_i = \mathit{InitialMethod}()}{\mathit{InitialLT} = \mathit{LT}[(\texttt{C}_0, \texttt{K}) : \kappa_0, (\texttt{C}_0, \bar{\texttt{M}}_0) : \bar{\kappa}_0, (\texttt{C}_1, \texttt{K}) : \kappa_1, (\texttt{C}_1, \bar{\texttt{M}}_1) : \bar{\kappa}_1, \ldots]}$$

Constructor Typing:

$$\frac{\begin{array}{c}\texttt{C}_0 = \texttt{class C extends D } \{\bar{\texttt{C}}\, \bar{\texttt{f}}; \texttt{K}\, \bar{\texttt{M}}\} \qquad \texttt{K} = \texttt{C}(\bar{\texttt{C}}\, \bar{\texttt{x}})\ \{\texttt{super}(\bar{\texttt{e}}); \bar{\texttt{s}}\} \qquad \texttt{D} \neq \texttt{Object}\\ \mathit{InitialLT}(\texttt{C}_0, \texttt{K}) : \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \emptyset \qquad \Gamma[\bar{\texttt{x}} : \bar{t}, \texttt{this} : t_t], s_p, i_p, \emptyset \vdash \bar{\texttt{e}} : \bar{\tau} \backslash \bar{\mathcal{C}}\\ \Gamma[\bar{\texttt{x}} : \bar{t}, \texttt{this} : t_t], s_p, i_p \vdash \bar{\texttt{s}} : \tau \backslash \mathcal{C} \qquad \bar{t'} = \mathit{FreeTypeVar}(\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} \cup \mathcal{C}' \cup \{\texttt{D.K}(\bar{\tau}, t_t \xrightarrow{s_p, i_p} t_r)\})\end{array}}{\mathit{InitialLT} \vdash_M (\texttt{C}_0, \texttt{K}) : \forall \bar{t'}. \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} \cup \bar{\mathcal{C}} \cup \{\texttt{D.K}(\bar{\tau}, t_t \xrightarrow{s_p, i_p} t_r)\} \cup \{\tau <: t_r\}}$$

$$\frac{\begin{array}{c}\texttt{C}_0 = \texttt{class C extends Object } \{\bar{\texttt{C}}\, \bar{\texttt{f}}; \texttt{K}\, \bar{\texttt{M}}\} \qquad \texttt{K} = \texttt{C}(\bar{\texttt{C}}\, \bar{\texttt{x}})\ \{\texttt{super}(); \bar{\texttt{s}}\}\\ \mathit{InitialLT}(\texttt{C}_0, \texttt{K}) : \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \emptyset \qquad \Gamma[\bar{\texttt{x}} : \bar{t}, \texttt{this} : t_t], s_p, i_p \vdash \bar{\texttt{s}} : \tau \backslash \mathcal{C} \qquad \bar{t'} = \mathit{FreeTypeVar}(\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C})\end{array}}{\mathit{InitialLT} \vdash_M (\texttt{C}_0, \texttt{K}) : \forall \bar{t'}. \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\}}$$

Method Typing:

$$\frac{\begin{array}{c}\texttt{C}_0 = \texttt{class C extends D } \{\bar{\texttt{C}}\, \bar{\texttt{f}}; \texttt{K}\, \bar{\texttt{M}}\} \qquad \texttt{M} = \texttt{RT m}(\bar{\texttt{C}}\, \bar{\texttt{x}})\ \{\bar{\texttt{s}}\} \qquad \mathit{InitialLT}(\texttt{C}_0, \texttt{M}) : \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \emptyset\\ \Gamma[\bar{\texttt{x}} : \bar{t}, \texttt{this} : t_t], s_p, i_p \vdash \bar{\texttt{s}} : \tau \backslash \mathcal{C} \qquad \bar{t'} = \mathit{FreeTypeVar}(\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\})\end{array}}{\mathit{InitialLT} \vdash_M (\texttt{C}_0, \texttt{M}) : \forall \bar{t'}. \bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C} \cup \{\tau <: t_r\}}$$

Class Typing:

$$\frac{\begin{array}{c}\mathit{InitialLT} \vdash_M (\texttt{C}_0, \texttt{K}) : \kappa_0 \qquad \mathit{InitialLT} \vdash_M (\texttt{C}_0, \bar{\texttt{M}}_0) : \bar{\kappa}_0\\ \mathit{InitialLT} \vdash_M (\texttt{C}_1, \texttt{K}) : \kappa_1 \qquad \mathit{InitialLT} \vdash_M (\texttt{C}_1, \bar{\texttt{M}}_1) : \bar{\kappa}_1 \qquad \ldots\end{array}}{\vdash_C \mathit{LT}[(\texttt{C}_0, \texttt{K}) : \kappa_0, (\texttt{C}_0, \bar{\texttt{M}}_0) : \bar{\kappa}_0, (\texttt{C}_1, \texttt{K}) : \kappa_1, (\texttt{C}_1, \bar{\texttt{M}}_1) : \bar{\kappa}_1, \ldots]}$$

Program Typing:

$$\frac{\begin{array}{c}\vdash_C \mathit{LT}[(\texttt{C}_0, \texttt{K}) : \kappa_0, (\texttt{C}_0, \bar{\texttt{M}}_0) : \bar{\kappa}_0, (\texttt{C}_1, \texttt{K}) : \kappa_1, (\texttt{C}_1, \bar{\texttt{M}}_1) : \bar{\kappa}_1, \ldots]\\ \emptyset, \emptyset, u \vdash \bar{\texttt{s}} : \tau \backslash \mathcal{C} \qquad \mathit{Closure}(\mathit{LT}[(\texttt{C}_0, \bar{\texttt{M}}_0) : \bar{\kappa}_0, (\texttt{C}_1, \bar{\texttt{M}}_1) : \bar{\kappa}_1, \ldots], \mathcal{C}) \text{ is consistent}\end{array}}{\vdash_P \{\texttt{C}_0, \texttt{C}_1, \ldots\};\ \bar{\texttt{s}} : \tau \backslash \mathcal{C}}$$

Fields:

$$\frac{}{\mathit{fields}(\texttt{Object}) = \emptyset} \qquad \frac{}{\mathit{fields}(\mathit{constants}) = \emptyset} \qquad \frac{\mathit{CT}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{C}}\, \bar{\texttt{f}}; \texttt{K}\, \bar{\texttt{M}}\} \qquad \mathit{fields}(\texttt{D}) = \bar{\texttt{D}}\, \bar{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \bar{\texttt{D}}\, \bar{\texttt{g}}, \bar{\texttt{C}}\, \bar{\texttt{f}}}$$

Figure 5: Label Type Rules for Classes and Programs

3.2.4 Class and Program Typing

Type inference rules for typing programs, classes, and methods are found in Figure 5. Programs are typed by typing each class definition, which types each method definition, which are in turn typed according to the expression rules in Figure 3. $\bar{\texttt{s}}$, representing `main` is also typed. Notice the initial integrity program counter must be the highest integrity label, so as not to

Closure Rules:

$$\frac{\tau <: \textbf{get}\ t}{\tau <: t}\text{(Get)} \qquad \frac{\tau <: \textbf{set}\ \tau'}{\tau' <: \tau}\text{(Set)} \qquad \frac{\mathcal{S}_1 <: \mathsf{s} \qquad \mathcal{S}_2 \cup (\mathsf{s} - \mathcal{S}_3) <: \mathcal{S}_4}{\mathcal{S}_2 \cup (\mathcal{S}_1 - \mathcal{S}_3) <: \mathcal{S}_4}(\mathcal{S}\text{-Trans}) \qquad \frac{\mathcal{S}_1 \cup \mathcal{S}_2 <: \mathcal{S}_3}{\mathcal{S}_1 <: \mathcal{S}_3 \qquad \mathcal{S}_2 <: \mathcal{S}_3}(\mathcal{S}\text{-Union})$$

$$\frac{\mathcal{I}_1 \cap \mathcal{I}_2 <: \mathcal{I}_3}{\mathcal{I}_1 <: \mathcal{I}_3 \qquad \mathcal{I}_2 <: \mathcal{I}_3}(\mathcal{I}\text{-Intersect}) \qquad \frac{\mathcal{I}_1 <: \mathsf{i} \qquad \mathsf{i} \cap \mathcal{I}_2 <: \mathcal{I}_3}{\mathcal{I}_1 \cap \mathcal{I}_2 <: \mathcal{I}_3}(\mathcal{I}\text{-Trans}) \qquad \frac{\mathcal{F}_1 <: \mathsf{f} \qquad \mathsf{f} <: \mathcal{F}_2}{\mathcal{F}_1 <: \mathcal{F}_2}(\mathcal{F}\text{-Trans})$$

$$\frac{\begin{array}{c}\mathcal{F} <: \mathsf{f} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ \mathcal{S}_1 \cup (\mathsf{f}.\mathtt{f}.\mathsf{S} - \mathcal{S}_2) <: \mathcal{S}_3\end{array}}{\mathcal{S}_1 \cup (\mathcal{F}.\mathtt{f}.\mathsf{S} - \mathcal{S}_2) <: \mathcal{S}_3}(\mathcal{S}\text{-Field}) \qquad \frac{\begin{array}{c}\mathcal{F} <: \mathsf{f} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ \mathcal{I}_1 \cap \mathsf{f}.\mathtt{f}.\mathsf{I} <: \mathcal{I}_2\end{array}}{\mathcal{I}_1 \cap \mathcal{F}.\mathtt{f}.\mathsf{I} <: \mathcal{I}_2}(\mathcal{I}\text{-Field}) \qquad \frac{\begin{array}{c}\mathcal{F} <: \mathsf{f} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ \mathsf{f}.\mathtt{f}.\mathsf{F} <: \mathcal{F}_1\end{array}}{\mathcal{F}.\mathtt{f}.\mathsf{F} <: \mathcal{F}_1}(\mathcal{F}\text{-Field})$$

$$\frac{\mathcal{F} <: \mathsf{f} \qquad \mathsf{f}.\mathtt{f}.\mathsf{A} <: \mathcal{A}_1 \qquad \mathcal{F} \text{ contains } \mathtt{f}}{\mathcal{F}.\mathtt{f}.\mathsf{A} <: \mathcal{A}_1}(\mathcal{A}\text{-Field}) \qquad \frac{\begin{array}{c}\mathsf{f} <: \mathcal{F} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ \mathcal{S} <: \mathsf{f}.\mathtt{f}.\mathsf{S}\end{array}}{\mathcal{S} <: \mathcal{F}.\mathtt{f}.\mathsf{S}}(\mathcal{S}\text{-Field'})$$

$$\frac{\begin{array}{c}\mathsf{f} <: \mathcal{F} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ \mathcal{I} <: \mathsf{f}.\mathtt{f}.\mathsf{I}\end{array}}{\mathcal{I} <: \mathcal{F}.\mathtt{f}.\mathsf{I}}(\mathcal{I}\text{-Field'}) \qquad \frac{\begin{array}{c}\mathsf{f} <: \mathcal{F} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ \mathcal{F}' <: \mathsf{f}.\mathtt{f}.\mathsf{F}\end{array}}{\mathcal{F}' <: \mathcal{F}.\mathtt{f}.\mathsf{F}}(\mathcal{F}\text{-Field'}) \qquad \frac{\begin{array}{c}\mathsf{f} <: \mathcal{F} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ \mathcal{A} <: \mathsf{f}.\mathtt{f}.\mathsf{A}\end{array}}{\mathcal{A} <: \mathcal{F}.\mathtt{f}.\mathsf{A}}(\mathcal{A}\text{-Field'})$$

$$\frac{\mathcal{A}_1 <: \mathsf{i} \qquad \mathsf{i} <: \mathcal{A}_2}{\mathcal{A}_1 <: \mathcal{A}_2}(\mathcal{A}\text{-Trans})$$

$$\frac{\begin{array}{c}\mathtt{C} <: \mathcal{A} \qquad \mathcal{A}.\mathtt{m}(\bar{\tau}, \tau_t \xrightarrow{pc, pc_i} \tau_r) \qquad mtype(\mathtt{C}, \mathtt{m}) = \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C}\\ \bar{\tau} = \overline{\langle\, \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A}\,\rangle} \qquad \tau_t = \langle\, \mathcal{S}_t, \mathcal{I}_t, \mathcal{F}_t, \mathcal{A}_t\,\rangle \qquad \tau_r = \langle\, \mathcal{S}_r, \mathcal{I}_r, \mathcal{F}_r, \mathcal{A}_r\,\rangle \qquad \bar{t''} = \theta(\bar{t'}, \mathtt{C}, \mathtt{m}, \bar{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r)\end{array}}{[\bar{t'} \mapsto \bar{t''}][s_p \mapsto pc, i_p \mapsto pc_i, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau_t, t_r \mapsto \tau_r]\mathcal{C}}(Method)$$

$$\frac{SC(\mathtt{L}, (\mathsf{s} \cup \mathcal{S}_2) - \mathcal{S}_3) \qquad \mathcal{S}_1 <: \mathsf{s}}{SC(\mathtt{L}, (\mathcal{S}_1 \cup \mathcal{S}_2) - \mathcal{S}_3)}\text{(SC-Trans)} \qquad \frac{IC(\mathtt{L}, \mathcal{I}_2 \cap \mathsf{i}) \qquad \mathcal{I}_1 <: \mathsf{i}}{IC(\mathtt{L}, \mathcal{I}_2 \cap \mathcal{I}_1)}\text{(IC-Trans)}$$

$$\frac{\begin{array}{c}\mathcal{F} <: \mathsf{f} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ SC(\mathtt{L}, (\mathcal{S} \cup \mathsf{f}.\mathtt{f}.\mathsf{S}) - \mathcal{S}')\end{array}}{SC(\mathtt{L}, (\mathcal{S} \cup \mathcal{F}.\mathtt{f}.\mathsf{S}) - \mathcal{S}')}\text{(SC-Field)} \qquad \frac{\begin{array}{c}\mathcal{F} <: \mathsf{f} \qquad \mathcal{F} \text{ contains } \mathtt{f}\\ IC(\mathtt{L}, \mathcal{I} \cap \mathsf{f}.\mathtt{f}.\mathsf{I})\end{array}}{IC(\mathtt{L}, \mathcal{I} \cap \mathcal{F}.\mathtt{f}.\mathsf{I})}\text{(IC-Field)}$$

Auxiliary Definitions:

$$\frac{LT(\mathtt{C}, \mathtt{m}) = \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C}}{mtype(\mathtt{C}, \mathtt{m}) = \forall \bar{t'}.\bar{t}, t_t \xrightarrow{s_p, i_p} t_r \backslash \mathcal{C}} \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{m} \text{ is not defined in } \bar{\mathtt{M}}}{mtype(\mathtt{C}, \mathtt{m}) = mtype(\mathtt{D}, \mathtt{m})}$$

$$\theta(t, \mathtt{C}, \mathtt{m}, \bar{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r) = t^{\mathtt{C}, \mathtt{m}, \mathit{flatten}(\bar{\mathcal{A}}, \mathcal{A}_t, \mathcal{A}_r)} \qquad \mathit{flatten}(\mathcal{A}^{\sigma}) = \mathcal{A} \qquad \mathit{flatten}(x, y, \ldots) = \mathit{flatten}(x), \mathit{flatten}(y), \ldots$$

Figure 6: Label Closure Rules and Definitions

unnecessarily reduce the integrity of any information. Methods require the type variables to be set in an initial label table in order to support recursive class definitions and mutually recursive methods. Method typing fills in the constraint types in the full label table, where the return type of the method body flows into the return label variable of the method. As previously noted, methods and constructors are given $\forall$ types so that they may vary polymophically, and these types are instantiated when computing the constraint closure.

3.2.5 Label Closure

The key closure rules for label constraint sets are given in Figure 6, along with some necessary definitions. Most of the rules add new constraints based on transitivity, obvious set propagations, and field labels. The closure rule (Method) is important for tying up the types of method calls. As discussed above, method constraints are added during method invocation, when the actual class of the object on which the method is being called may be unknown. Thus, for all constraints $\mathtt{C} <: \mathcal{A}$, where C is a concrete class, the method m is looked up in LT via *mtype*, which returns a typing for that method as found either in C or in a superclass if not defined in C. We then substitute the labels in the *method* constraint into this constraint set from the label table, and replace all local label variables as defined by the function θ.

The manner in which local label variables are replaced defines the *contours* of a concrete class analysis. In other words, different instantiations of the $\forall$ type create unique types that distinguish different method invocations. Our definition of θ creates a new contour for each distinct receiver type C, method name f, argument type $\bar{\mathcal{A}}$ ($\mathcal{A}_t$ is the type of this), and return type $\mathcal{A}_r$. This allows calls to be distinguished based on receiver and argument types, as in CPA [3], and additionally distinguishes call-sites based on unique program points. Since the (Invoke) type rule creates fresh variables for each method invocation, this serves as a unique marker of the call-site in the program; thus, $\mathcal{A}_r$ is the call-site of the method. Since constructor calls during (New) are similar to method invocations, the analysis can distinguish most object instances via call-sites and constructor arguments. Consider the following example.

$\mathtt{x} = \mathtt{new\ C}(); \mathtt{y} = \mathtt{new\ C}();$
$\mathtt{x.put}(\mathtt{read}_{(\{\mathtt{L}\},\{\mathtt{L}\})}(\mathtt{fd})); \mathtt{y.put}(\mathtt{read}_{(\{\mathtt{H}\},\{\mathtt{H}\})}(\mathtt{fd}'))$
$\mathtt{x.get}();$

Here, our analysis produces separate contours for the creation of x and y, where CPA merges them into one. Even though the put calls have different contours, since the types of x and y are not distinguished, the CPA analysis cannot determine that $\mathtt{x.get}()$ is low. We obtain more precision, so we can correctly identify the flows of data into and out-of abstract objects on the heap.

This precision is similar to that obtained in data-polymorphic CPA analysis [27, 33]; although DCPA includes many optimizations to combine contours whenever possible, while still supporting data polymorphism. The *flatten* function is necessary to merge contours for recursive calls and to ensure the analysis terminates. We discuss the termination of this algorithm in section 3.2.7.

We define a constraint closure as follows.

Definition 3.1 (Constraint Closure) *$Closure(LT, \mathcal{C})$ is the least set that includes $\mathcal{C}$ and any constraint that can be derived from $\mathcal{C}$ by the rules of Figure 6, and with the additional constraint that the (Method) rule is only applied once in the closure for each unique set of premises.*

If we did not constrain (*Method*) rule as above, it could be applied arbitrarily many times, generating different fresh variables each time.

3.2.6 Inconsistent Constraints

Inconsistencies in the label constraint sets come from SC and IC constraints. Constraint consistency is defined as follows.

Definition 3.2 (Inconsistent Constraints) *An inconsistent constraint is any constraint $SC(\mathtt{L}_s, \mathtt{L}'_s)$, where $\mathtt{L}'_s \not\subseteq \mathtt{L}_s$; or any constraint $IC(\mathtt{L}_i, \mathtt{L}'_i)$, where $\mathtt{L}_i \not\subseteq \mathtt{L}'_i$.*

Note that constraint consistency is defined only on concrete constraint sets, which are formed during the closure after all transitive flows into type variables have been considered. If $Closure(LT, \mathcal{C})$ contains an inconsistent constraint, then the closure is inconsistent, and type inference fails.

Secrecy policies are enforced by SC constraints. In the constraint $SC(\mathtt{L}_s, \mathtt{L}'_s)$, $\mathtt{L}_s$ is the secrecy policy of the IO channel, and $\mathtt{L}'_s$ is the set of labels on the data at that point. Proper enforcement of the policy requires the labels on the data to be a subset of the labels on the IO channel. For example, the constraint $SC(\{\mathtt{high}, \mathtt{low}\}, \{\mathtt{low}\})$ is consistent, with low data flowing to a high channel; $SC(\{\mathtt{low}\}, \{\mathtt{high}\})$ is inconsistent, since high data is flowing to a low channel.

Similarly, integrity policies are enforced by IC constraints. In $IC(\mathtt{L}_i, \mathtt{L}'_i)$, $\mathtt{L}_i$ is the integrity policy of the IO channel, and $\mathtt{L}'_i$ is the set of labels on the data at that point. Since integrity is a dual to secrecy, the subset relation is flipped, meaning $\mathtt{L}_i \subseteq \mathtt{L}'_i$ is required to satisfy the policy. For example, the constraint $IC(\{\mathtt{Untainted}\}, \{\mathtt{Untainted}, \mathtt{Classified}\})$ is consistent, as the data is required to carry at least the untainted label; whereas $IC(\{\mathtt{Untainted}, \mathtt{Classified}\}, \{\mathtt{Untainted}\})$ is inconsistent, since the data must be both untainted and classified.

3.2.7. Typing Complexity and Termination

A potential pitfall of this form of type inference algorithm is non-termination. If contours are continually created for recursive method invocations, the analysis may not terminate. Our analysis merges contours for recursive calls, ensuring termination. We now address the complexity of type inference and constraint closure computation.

Inferring types completes in linear time. Closing the constraint set can be exponential in the worst case. This is evident from the definition of θ. t inputs to θ are all flat (i.e. have no superscript), since they are the free type variables that occur when typing method `m` of class `C`. Superscripted variables are only added during the closure. This means t is bounded by n, the size of the program. Since $\bar{\mathcal{A}}, \mathcal{A}_t$, and $\mathcal{A}_r$ are all flattened, the number of possibilities for these values is bounded by the number of concrete classes and the number of fresh variables created in the program, which are each less than n. Thus, in the worst case, we may create up to n^{n^5} contours (accounting also for `C` and `m`). This is a large exponential, but nevertheless terminates. Many optimizations (*e.g.* combining contours and constraint garbage collection) can be performed to make this practical, as shown in [27, 33] and elsewhere; this is out of the scope of the current work.

The type inference system provides separate compilation of classes, since type inference can be done separately, and the final global constraint set must be closed and checked for inconsistencies. Classes and methods may be analyzed only once, and their types and constraints built into the label table, which may be re-used for any number of programs.

4 Soundness and Noninterference

We now state the formal soundness and noninterference properties for our system. Soundness means that well-typed programs will not produce any run-time secrecy or integrity check failures. Noninterference is shown here only for secrecy: changes made to high inputs do not effect low outputs. We first provide an overview of the proof technique, which is a new method for proving noninterference using a labeled operational semantics, and state the results.

In order to more clearly state our results, we make the following assumptions in the definitions and proofs in this section. The program has a fixed, well-typed class table, CT, and a fixed, well-typed label table, LT. We use $\texttt{high} \in \bar{S}$ to represent $\forall S_i \in \bar{S}, \texttt{high} \in S_i$. In order to simplify our presentation, integrity labels have generally been omitted. While this does affect the soundness theorem concerning integrity checks, extending the proofs to integrity labels is straight-forward. For the purposes of proving noninterference, integrity labels and types are irrelevant; however, the dual property of *integrity noninterference*, where low integrity inputs do not affect high integrity outputs can be shown, with an identical proof to secrecy noninterference. In proving noninterference, we assume expressions and statements do not contain any $\texttt{Declassify}(\texttt{e}', \texttt{L})$ subexpressions, which would violate the property that *high* inputs do not affect *low* outputs. Hence, in the proof of noninterference, we assume `Declassify` does not occur in any programs. This restriction does not apply to the soundness result, wherein `Declassify` may occur in programs, and run-time check failures will not occur; these programs may, however, permit interfering executions.

Our Noninterference property, Theorem 4.24, states that for a typeable program `P`, any two runs of the program differing only in high input streams will produce the same low output streams, and that the resulting low input streams are also equivalent. The latter condition is necessary since the size of the low input streams after computation may convey secret information, such as if one stream had been read five times, and the other seven; the attacker would know that a change was made to a high input. We specify that the values must be integers for this theorem, as it intuitively doesn't make sense to input or output heap locations (pointers). Since our system is termination-insensitive, both runs of the program are assumed to terminate normally.

The proof of noninterference proceeds as follows. We define labeled configurations that are analogous to the definitions of expressions and statements. We then translate the original program into a configuration, mapping the labels given by the typing of each sub-expression onto each sub-configuration. This creates a one-to-one correspondence between the expression types and semantic labels.

We define a small-step operational semantics, where computation is specified on four-tuples: configurations, heaps, sets of input streams and sets of output streams. As expected, heaps consist of labeled objects and fields. Input and output streams are represented by file descriptors, and the run-time streams also include policies. These policies must be checked against the static policies of read and write commands when they are used. If we did not perform this check, we would allow multiple security access levels to the same channel. This would mean that the security level on the channel is dynamically changing, which is not supported by our system. Our definition of noninterference would not hold under these conditions, since a low observer may have intermittent access to a channel that also uses high data. We assume in our proofs that the static policy description of an IO command aligns with the run-time policy of the channel being used. Since the stream policy is a purely dynamic property, it is impossible to verify statically.

The first step in our formal analysis is a proof of a Fixed Point Lemma, which assures that during computation the labels on configurations and the labels on the heap will never increase. This is analogous to a subject-reduction lemma, since the types have been mapped onto the semantics. Using this Fixed Point Lemma, we then show soundness of the type system that well-typed programs do not produce run-time check failures.

For proving noninterference, we first distinguish low and high input and output channels. The observational behavior of the low user is defined by a set of secrecy labels `Low`. Hence, low data is any data labeled with some set of labels `L`, such that $\mathtt{L} \subseteq \mathtt{Low}$. High data is any data not observable to the low user, e.g. data labeled with a set of secrecy labels `H`, such that $\mathtt{H} \not\subseteq \mathtt{Low}$.

Noninterference is then shown via bisimulation of the execution of two configurations. The bisimulation relation, $\simeq_{\mathtt{Low}}$, states that high labeled values may differ, and any low values in the configurations must be equivalent. Furthermore, the data labeled low in the heaps must exist in the other heap with the same values; high portions of the heap are not accounted by the bisimulation. The final part of the bisimulation asserts the equivalence of low input and output streams.

We distinguish low reduction steps from high reduction steps based on the labels of the configuration, such that every reduction step is either a low step or a high step. We then show that for typeable programs, assuming two executions where the low input streams are identical, they each take the same low steps, with possibly differing high steps between, such that after each low step and number of high steps, the resulting configurations remain bisimilar. Hence, when the execution finishes, the result is a low-equivalent trace of inputs and outputs.

In order to cleanly state our result, we define an unlabeled semantics that works directly on expressions (and statements). This semantics is equivalent to the configuration semantics, only lacking labels. The noninterference result on this semantics is the same as that of the labeled semantics: that if a program is typeable, for two terminating executions of the program that differ only in high input streams, the resulting low input and output streams are equivalent (as are the termination values of the executions).

The structure of the remainder of this section is as follows. Section 4.1 provides some necessary definitions, and Section 4.2 defines a labeled small-step operational semantics. The Fixed Point Lemma and Soundness theorem appear in Section 4.3, and the Noninterference result is in Section 4.4.

4.1 Definitions

We define concrete secrecy types of expressions and statements in Definition 4.1 so as to more cleanly state our results. Concrete secrecy types are the concrete secrecy labels for an expression as determined by the type of the expression and the constraint set.

Definition 4.1 (Concrete Secrecy Types) $\Gamma, H, pc, \mathcal{C} \vdash_{con} \epsilon : S$ *is an assertion and* S *is a concrete secrecy type as follows. For some* $pc_i, \mathcal{I}, \mathcal{F}, \mathcal{A}$*, if* $\Gamma, pc, pc_i, H \vdash \mathtt{e} : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}'$ *or* $\Gamma, pc, pc_i, H \vdash \mathtt{s} : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}'$ *where* $\mathcal{C}' \subseteq \mathcal{C}$*,* S *is a set containing every concrete label,* `l`*, such that either*

1. $\mathtt{l} \in \mathcal{S}$*; or*
2. *there exists an* $s \in \mathcal{S}$*, such that* $\mathtt{l} <: s \in \mathit{Closure}(LT, \mathcal{C})$*; or*
3. *there exists an* $f.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathcal{S}$*, such that* $\mathtt{l} <: f.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathit{Closure}(LT, \mathcal{C})$

The translation function $[\![\Gamma, H_i, pc, \mathcal{C}, \epsilon]\!] = (\!| \mathcal{E} |\!)^S, H$ is given in Definition 4.4. For every subexpression ϵ' of ϵ, the concrete label type of ϵ' is placed as the label on the corresponding subconfiguration in $(\!| \mathcal{E} |\!)^S$. Thus, there is a one-to-one correspondence between concrete label types of subexpressions of ϵ, and the labels on each subconfiguration in $(\!| \mathcal{E} |\!)^S$. The heap is also translated, as defined in Definition 4.2, such that any constraint in $\mathcal{C}$ that has a concrete label flowing into a heap location is placed on the translated heap.

Definition 4.2 (Heap Translation) $[\![\Gamma, H_i, \mathcal{C}]\!] = H$*, iff* $\forall o \in H_i$*, if* $H_i(o) = (\!| \mathtt{new}\ \mathtt{C}(\overline{(\!| v |\!)^{S_v}}) |\!)^S$*,* $\Gamma(o) = \bar{t}$*, and* $\bar{S} <: \bar{t} \in \mathcal{C}$*, then* $H(o) = H_i[o \mapsto (\!| \mathtt{new}\ \mathtt{C}(\overline{(\!| v |\!)^{S_v \cup S}}) |\!)^S]$

Definition 4.3 (Expression and Statement Translation) *If* $\Gamma, H, pc, \mathcal{C} \vdash_{con} \epsilon : S$*, the translation* $[\![\Gamma, H, pc, \mathcal{C}, \epsilon]\!]_e = (\!| \mathcal{E} |\!)^S$ *and is defined by straightforward structural induction of* ϵ*. The base cases are* $[\![\Gamma, H, pc, \mathcal{C}, \mathtt{c}]\!]_e = (\!| \mathtt{c} |\!)^S$*,* $[\![\Gamma, H, pc, \mathcal{C}, \mathtt{x}]\!]_e = (\!| \mathtt{x} |\!)^S$*,* $[\![\Gamma, H, pc, \mathcal{C}, (\!| v |\!)^S]\!]_e = (\!| v |\!)^S$*, and* $[\![\Gamma, H, pc, \mathcal{C}, (\!| \mathtt{read}_{\mathtt{L}}() |\!)^S]\!]_e = (\!| \mathtt{read}_{\mathtt{L}}() |\!)^S$*.* $\mathtt{new}\ (\bar{\mathtt{e}})$ *is a* <u>*special case*</u>*, defined as follows:* $[\![\Gamma, H, pc, \mathcal{C}, \mathtt{new}\ (\bar{\mathtt{e}})]\!] = (\!| \mathtt{new}\ \mathtt{C}([\![\Gamma, H, pc, \mathcal{C}, \bar{\mathtt{e}}]\!]_e)^{\bar{S}'} |\!)^S$ *iff* $\Gamma, pc, H \vdash \mathtt{new}\ (\bar{\mathtt{e}}) : \langle \mathcal{S}, \{\bar{\mathtt{f}} : \overline{\langle s, f, \alpha \rangle}\}, \mathcal{A} \rangle \backslash \mathcal{C}'$ *and for all* $\bar{S}' <: \bar{s} \in \mathcal{C}$*.*

For example, $[\![\Gamma, H, pc, \mathcal{C}, \mathtt{e} \oplus \mathtt{e}']\!]_e = (\!| [\![\Gamma, H, pc, \mathcal{C}, \mathtt{e}]\!]_e \oplus [\![\Gamma, H, pc, \mathcal{C}, \mathtt{e}]\!]_e |\!)^S$*.*

Concrete Labels:
$S, I ::= \{\bar{\mathtt{l}}\}$, where $\bar{\mathtt{l}}$ are unique label names
Heap Objects:
$ho ::= \llparenthesis \mathtt{new\ C}, \{\bar{\mathcal{V}}\} \rrparenthesis^S_I$
Heap:
H is a finite partial function from memory locations to heap objects.
Object Identifier:
loc is a unique memory location in the heap
$o ::= loc^{(S,I)}$ is an object identifier consisting of a memory location with it's security level.
IO Streams:
Functions from file descriptors with pairs of security levels to streams of integers.
$\iota ::= (\mathtt{fd}, S, I) \longrightarrow \bar{\mathtt{c}}$
$\omega ::= (\mathtt{fd}, S, I) \longrightarrow \bar{\mathtt{c}}$
Values:
$v ::= \mathtt{CO} \mid o \mid \mathit{CkFail} \mid \mathit{IOErr}$

Final Configurations:
$\mathcal{V} ::= \llparenthesis o \rrparenthesis^S_I \mid \llparenthesis \mathtt{c} \rrparenthesis^S_I \mid \mathit{CkFail} \mid \mathit{IOErr}$
Expressions:
$\mathtt{e} = \ldots \mid \mathcal{V} \mid \llparenthesis o \rrparenthesis^S_I.\mathtt{super}(\mathtt{C}, \bar{\mathtt{e}})$
Expr./Stmt.
$\epsilon = \mathtt{e} \mid \mathtt{s}$
Configurations:
$\mathcal{E} ::= v \mid \mathtt{x} \mid \mathtt{this} \mid \llparenthesis \mathcal{E} \rrparenthesis^S_I.\mathtt{f} \mid (\mathtt{C})\ \llparenthesis \mathcal{E} \rrparenthesis^S_I \mid$
$\llparenthesis \mathcal{E} \rrparenthesis^S_I \oplus \llparenthesis \mathcal{E} \rrparenthesis^{S'}_{I'} \mid \mathtt{new\ C}(\overline{\llparenthesis \mathcal{E} \rrparenthesis^S_I})^{\bar{S}'}_{\bar{I}'} \mid \llparenthesis \mathcal{E} \rrparenthesis^S_I.\mathtt{m}(\overline{\llparenthesis \mathcal{E} \rrparenthesis^S_I}) \mid$
$\mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathcal{E} \rrparenthesis^S_I) \mid \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathcal{E} \rrparenthesis^S_I, \llparenthesis \mathcal{E} \rrparenthesis^{S'}_{I'}) \mid$
$\mathtt{Declassify}(\llparenthesis \mathcal{E} \rrparenthesis^S_I, \mathtt{L}) \mid$
$\mathtt{if}\ \llparenthesis \mathcal{E} \rrparenthesis^S_I\ \mathtt{then}\ \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}\ \mathtt{else}\ \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \mid$
$\overline{\llparenthesis \mathcal{E} \rrparenthesis^S_I} \mid \{\llparenthesis \mathcal{E} \rrparenthesis^S_I\} \mid \llparenthesis \mathcal{E} \rrparenthesis^S_I.\mathtt{f} := \llparenthesis \mathcal{E}' \rrparenthesis^{S'}_{I'}; \mid$
$\mathtt{return}\ \llparenthesis \mathcal{E} \rrparenthesis^S_I; \mid \llparenthesis o \rrparenthesis^S_I.\mathtt{super}(\overline{\llparenthesis \mathcal{E} \rrparenthesis^S_I})|;$
Reductions:
$\llparenthesis \mathcal{E} \rrparenthesis^S_I, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E}' \rrparenthesis^{S'}_{I'}, H', \iota', \omega'$

Figure 7: Operational Semantics Definitions

Definition 4.4 (Translation) $[\![\Gamma, H_i, pc, \mathcal{C}, \epsilon]\!] = \llparenthesis \mathcal{E} \rrparenthesis^S, H$ *iff* $[\![\Gamma, H_i, \mathcal{C}]\!] = H$ *and* $[\![\Gamma, H, pc, \mathcal{C}, \epsilon]\!]_e = \llparenthesis \mathcal{E} \rrparenthesis^S$

4.2 Semantics

We now present a small-step operational semantics for our system. Figure 7 shows the necessary definitions for the semantics. Configurations allow each sub-expression to be labeled. Reductions are then on labeled configurations, $\llparenthesis \mathcal{E} \rrparenthesis^S_I$, with two label sets S and I, for secrecy and integrity, respectively. Labels in the label sets may appear due to direct or indirect flows. Final configurations, $\llparenthesis v \rrparenthesis^S_I$, are values with an associated label set, or *CkFail*, denoting a failed label check, or *IOErr*, denoting a mismatch in read or write policies. Objects on the heap are labeled so we can separate the *low* and *high* portions of the heap for noninterference. Object identifiers must contain additional labels to distinguish between two different pointers to the same object. Thus, $\llparenthesis o \rrparenthesis^H_H$ and $\llparenthesis o \rrparenthesis^L_L$ point to the same object, but have different labels, due to computing in different contexts, with different program counters. ι is a function mapping file descriptors and sets of security labels to a list of integer input values. ω is a function mapping file descriptors and sets of security labels to a list of integer output values.

Small-step reduction rules define the single-step reduction relation $\llparenthesis \mathcal{E} \rrparenthesis^S_I, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E}' \rrparenthesis^{S'}_{I'}, H', \iota', \omega'$. The semantics is designed with a noninterference theorem in mind. As discussed previously, labels from the type system are mapped onto configurations in the semantics. Thus, a semantic program runs with all of the labels inferred by the type system. Notice the semantics are defined non-determinalistically, due to the use of the translations when invoking methods or calling constructors. Since the type rules used during evaluations may contain uses of (Sub), the translation becomes non-deterministic. The choice of the type variables added to Γ' in the (New-R) rule also allows non-determinism.

Reduction rules are given in Figure 8 and Figure 9. Reductions under context rules are in Figure 10. The reductions implicitly work over environments Γ and constraint sets $\mathcal{C}$, which are necessary for the method invocation and constructor reductions, which require a typing translation during the reduction.

4.3 Soundness and Fixed Point

We now show that for translated programs, there exists a reduction such that the labels on each configuration are a fixed-point; that is, the labels will never increase during computation. The existence of a reduction is due to the non-determinism in our semantics, as discussed above. This means that any configuration marked *low*, will never compute to a *high* value, which is necessary for noninterference. The fixed-point also allows us to state our soundness result that no typable programs will produce run-time check failures (although they may have run-time IO errors, where the stream policy does not align with the static stream label). We proceed by stating a few necessary lemmas before giving the results.

(Field-R) $\llparenthesis \llparenthesis o \rrparenthesis^{S'}_{I'}.\texttt{f}_i \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis v \rrparenthesis^{S_i \cup S \cup S'}_{I_i \cap I \cap I'}, H, \iota, \omega$ where $\mathit{fields}(\texttt{C}) = \bar{\texttt{C}}\,\bar{\texttt{f}}$, and $H(o) = \llparenthesis \texttt{new C}(\bar{\mathcal{V}}) \rrparenthesis^{S_v}_{I_v}$ and $\mathcal{V}_i = \llparenthesis v \rrparenthesis^{S_i}_{I_i}$

(Op-R) $\llparenthesis \llparenthesis \texttt{c} \rrparenthesis^{S_c}_{I_c} \oplus \llparenthesis \texttt{c}' \rrparenthesis^{S'_c}_{I'_c} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis v \rrparenthesis^{S_c \cup S'_c \cup S}_{I_c \cap I'_c \cap I}, H, \iota, \omega$ where $v = \texttt{c} \oplus \texttt{c}'$

(Cast-R) $\llparenthesis (\texttt{D})\ \llparenthesis o \rrparenthesis^{S_v}_{I_v} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis o \rrparenthesis^{S_v \cup S}_{I_v \cap I}, H, \iota, \omega$ where $\texttt{C} <: \texttt{D}$

(Declassify-R) $\llparenthesis \texttt{Declassify(}\llparenthesis v \rrparenthesis^{S_v}_{I_v}\texttt{,L)} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis v \rrparenthesis^{(S_v - \texttt{L}) \cup S}_{I_v \cap I}, H, \iota, \omega$

(New-R) $\llparenthesis \texttt{new C}(\llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}_{\bar{I_v}})^{\bar{S}}_{\bar{I}} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'}_{I'}; \llparenthesis \texttt{return}\ \llparenthesis o \rrparenthesis^{S}_{I}; \rrparenthesis^{S}_{I} \rrparenthesis^{S}_{I}, H', \iota, \omega$
where $\mathit{cnbody}(\texttt{C}) = \texttt{super}(\bar{\texttt{e}}); \bar{\texttt{s}}$ and $\texttt{class C extends D}\ \{\dots\}$ and $\Gamma' = \Gamma[o \mapsto \bar{t}]$
and $\llparenthesis \mathcal{E}' \rrparenthesis^{S'}_{I'} = \llbracket \Gamma', H', S, \mathcal{C}, [\bar{\texttt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}_{\bar{I_v}}, \texttt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}_{I_v}]\texttt{this.super}(\texttt{D}, \bar{\texttt{e}}); \bar{\texttt{s}} \rrbracket$
and $H' = H[o \mapsto \llparenthesis \texttt{new C}(\llparenthesis \overline{\texttt{null}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}]$and $o = \mathit{newref}(H, S, I)$

(Invoke-R) $\llparenthesis \llparenthesis o \rrparenthesis^{S_v}_{I_v}.\texttt{m}(\llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}_{\bar{I_v}}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E}' \rrparenthesis^{S' \cup S}_{I' \cap I}, H, \iota, \omega$
where $\mathit{mbody}(\texttt{m}, \texttt{C}) = \bar{\texttt{s}}$
and $\llparenthesis \mathcal{E}' \rrparenthesis^{S'}_{I'} = \llbracket \Gamma, H, S_v, \mathcal{C}, [\bar{\texttt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}_{\bar{I_v}}, \texttt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}_{I_v}]\bar{\texttt{s}} \rrbracket$

(Super-R) $\llparenthesis \llparenthesis o \rrparenthesis^{S_v}_{I_v}.\texttt{super}(\texttt{C}, \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}_{\bar{I_v}}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E}' \rrparenthesis^{S \cup S'}_{I \cap I'}, H, \iota, \omega$
where $\mathit{cnbody}(\texttt{C}) = \texttt{super}(\bar{\texttt{e}}); \bar{\texttt{s}}$ and $\texttt{class C extends D}\ \{\dots\}$
and $\llparenthesis \mathcal{E}' \rrparenthesis^{S'}_{I'} = \llbracket \Gamma, H, S_v, \mathcal{C}, [\bar{\texttt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}_{\bar{I_v}}, \texttt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}_{I_v}]\texttt{this.super}(\texttt{D}, \bar{\texttt{e}}); \bar{\texttt{s}} \rrbracket$

(Super-R') $\llparenthesis \llparenthesis o \rrparenthesis^{S_o}_{I_o}.\texttt{super(Object)} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \texttt{null} \rrparenthesis^{S \cup S_o}_{I \cap I_o}, H, \iota, \omega$

(IfTrue-R) $\llparenthesis \texttt{if}\ \llparenthesis \texttt{True} \rrparenthesis^{S_v}_{I_v} \texttt{then}\ \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}\ \texttt{else}\ \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1 \cup S}_{I_1 \cap I}, H, \iota, \omega$

(Seq-R) $\llparenthesis \llparenthesis v \rrparenthesis^{S_v}_{I_v}; \overline{\llparenthesis \mathcal{E} \rrparenthesis^{S}_{I}} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \overline{\llparenthesis \mathcal{E} \rrparenthesis^{S}_{I}} \rrparenthesis^{S}_{I}, H, \iota, \omega$ where $\overline{\llparenthesis \mathcal{E} \rrparenthesis^{S}_{I}}$ is a sequence two or more configs.

(Seq-R') $\llparenthesis \llparenthesis v \rrparenthesis^{S_v}_{I_v}; \llparenthesis \mathcal{E} \rrparenthesis^{S'}_{I'} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E} \rrparenthesis^{S' \cup S}_{I' \cap I}, H, \iota, \omega$

(Return-R) $\llparenthesis \texttt{return}\ \llparenthesis v \rrparenthesis^{S_v}_{I_v}; \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis v \rrparenthesis^{S_v \cup S}_{I_v \cap I}, H, \iota, \omega$

(Block-R) $\llparenthesis \{\llparenthesis \mathcal{E} \rrparenthesis^{S'}_{I'}\} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E} \rrparenthesis^{S \cup S'}_{I \cap I'}, H, \iota, \omega$

(Skip-R) $\llparenthesis ; \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \texttt{null} \rrparenthesis^{S}_{I}, H, \iota, \omega$

(Assign-R) $\llparenthesis \llparenthesis o \rrparenthesis^{S'}_{I'}.\texttt{f} := \llparenthesis v \rrparenthesis^{S_v}_{I_v}; \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \texttt{null} \rrparenthesis^{S \cup S_v \cup S'}_{I \cap I_v \cap I'},$
$H[o \mapsto \llparenthesis \texttt{new C}(\dots, \llparenthesis v \rrparenthesis^{S_v \cup S \cup S' \cup S_i}_{I_v \cap I \cap I' \cap I_i}, \dots) \rrparenthesis^{S''}_{I''}], \iota, \omega$
where $\mathit{fields}(\texttt{C}) = \bar{\texttt{C}}\,\bar{\texttt{f}}$ and $H(o) = \llparenthesis \texttt{new C}(\bar{\mathcal{V}}) \rrparenthesis^{S''}_{I''}$ and $\mathcal{V}_i = \llparenthesis v_i \rrparenthesis^{S_i}_{I_i}$, for i corresponding to $\texttt{f}$ in $\bar{\mathcal{V}}$

$\mathit{newref}(H, S, I) = o = \mathit{loc}^{(S,I)}_{i}$ where $i - 1$ is the largest integer, such that $\mathit{loc}^{(S,I)}_{i-1} \in H$

Figure 8: Operational Semantics Reduction Rules

Lemma 4.5 shows that the type of every expression must contain the respective secrecy and integrity program counters.

Lemma 4.5 (Pervasiveness of program counters)

1. *If* $\Gamma, pc, pc_i, H \vdash \epsilon : \langle \mathcal{S}, \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$, *then* $pc \subseteq \mathcal{S}$ *and* $pc_i \supseteq \mathcal{I}$.
2. *If* $\Gamma, H, pc, \mathcal{C} \vdash_{con} \epsilon : S$ *then* $pc \subseteq S$.

Proof.

1. By induction on the type derivation of $\texttt{e}$. For the base cases: Var, This, Const, New, Val, Heap, Label, Declassify, No-op the lemma holds with pc and pc_i added to the typing. The inductive step is trivial, since each rule unions the secrecy labels and intersects the integrity labels from the premise.
2. Directly by Lemma 4.5[1] and Definition 4.1.

(Input-R)	$\llparenthesis \texttt{read}_{(\texttt{L},\texttt{L}')}(\llparenthesis \texttt{fd} \rrparenthesis^{S_f}_{I_f}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \llparenthesis \texttt{c} \rrparenthesis^{\texttt{L}}_{\texttt{L}'}, H, \iota', \omega$	where $\texttt{L} = S_i$ and $\texttt{L}' = I_i$
		and $\iota(\texttt{fd}, S_i, I_i) = \texttt{c}.\iota'(\texttt{fd}, S_i, I_i)$
		and $S_f \subseteq \texttt{L}$ and $\texttt{L}' \subseteq I_f$
(Output-R)	$\llparenthesis \texttt{write}_{(\texttt{L},\texttt{L}')}(\llparenthesis \texttt{c} \rrparenthesis^{S_c}_{I_c}, \llparenthesis \texttt{fd} \rrparenthesis^{S_f}_{I_f}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow$	where $\texttt{L} = S_i$ and $\texttt{L}' = I_i$
	$\llparenthesis \texttt{null} \rrparenthesis^{S_c \cup S_f \cup S}_{I_c \cap I_f \cap I}, H, \iota, \omega'$	and $\omega'(\texttt{fd}, S_i, I_i) = \texttt{c}.\omega(\texttt{fd}, S_i, I_i)$
		and $S_c \cup S_f \subseteq \texttt{L}$ and $\texttt{L}' \subseteq I_c \cap I_f$
(InFail-R)	$\llparenthesis \texttt{read}_{(\texttt{L},\texttt{L}')}(\llparenthesis \texttt{fd} \rrparenthesis^{S_f}_{I_f}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \mathit{CkFail}, H, \iota, \omega$	where $\texttt{L} = S_i$ and $\texttt{L}' = I_i$
		and $\iota(\texttt{fd}, S_i, I_i)$
		and $S_f \not\subseteq \texttt{L}$ and $\texttt{L}' \not\subseteq I_f$
(OutFail-R)	$\llparenthesis \texttt{write}_{(\texttt{L},\texttt{L}')}(\llparenthesis \texttt{c} \rrparenthesis^{S_c}_{I_c}, \llparenthesis \texttt{fd} \rrparenthesis^{S_f}_{I_f}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \mathit{CkFail}, H, \iota, \omega$	where $\texttt{L} = S_i$ and $\texttt{L}' = I_i$
		and $\omega(\texttt{fd}, S_i, I_i)$
		and $S_c \cup S_f \not\subseteq \texttt{L}$ or $\texttt{L}' \not\subseteq I_c \cap I_f$
(InErr-R)	$\llparenthesis \texttt{read}_{(\texttt{L},\texttt{L}')}(\llparenthesis \texttt{fd} \rrparenthesis^{S_f}_{I_f}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \mathit{IOErr}, H, \iota, \omega$	where $\texttt{L} \neq S_i$ or $\texttt{L}' \neq I_i$
		and $\iota(\texttt{fd}, S_i, I_i)$
(OutErr-R)	$\llparenthesis \texttt{write}_{(\texttt{L},\texttt{L}')}(\llparenthesis \texttt{c} \rrparenthesis^{S_c}_{I_c}, \llparenthesis \texttt{fd} \rrparenthesis^{S_f}_{I_f}) \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \mathit{IOErr}, H, \iota, \omega$	where $\texttt{L} \neq S_i$ or $\texttt{L}' \neq I_i$
		and $\omega(\texttt{fd}, S_i, I_i)$
(CkFail-R)	$\llparenthesis \mathcal{E} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \mathit{CkFail}, H, \iota, \omega$	where CkFail is a subconf. of $\llparenthesis \mathcal{E} \rrparenthesis^{S}_{I}$
(IOErr-R)	$\llparenthesis \mathcal{E} \rrparenthesis^{S}_{I}, H, \iota, \omega \rightsquigarrow \mathit{IOErr}, H, \iota, \omega$	where IOErr is a subconf. of $\llparenthesis \mathcal{E} \rrparenthesis^{S}_{I}$

Figure 9: Operational Semantics IO Reduction Rules

□

Lemma 4.6 (PC Weakening)

1. *If* $\Gamma, pc, H \vdash \epsilon : \tau \setminus \mathcal{C}$ *and* $pc' \subseteq pc$*, then* $\Gamma, pc', H \vdash \epsilon : \tau \setminus \mathcal{C}$*.*
2. *If* $\Gamma, H, pc, \mathcal{C} \vdash_{con} \epsilon : S$ *and* $pc' \subseteq pc$*, then* $\Gamma, H, pc', \mathcal{C} \vdash \epsilon : S$*.*

Proof.

1. By induction on the derivation of ϵ, using the (Sub) rule.
2. Directly by Lemma 4.6[1] and Definition 4.1.

□

Fixed Point Lemma 4.9 shows that for any translated expression, taking a reduction step produces a configuration whose secrecy label is no larger than the given configuration.

Lemma 4.7 (Substitution) *If* $\Gamma[\bar{\texttt{x}} : \bar{t}], s_p, \emptyset \vdash \epsilon : \tau \setminus \mathcal{C}$*, and* $\Gamma, pc, H \vdash \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}} : \bar{\tau} \setminus \bar{\mathcal{C}}$*, and there exists* $\bar{t_l'}$ *such that for all* $\bar{t_l}$ *free in* $[s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}]\mathcal{C}$*,* $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}]\mathcal{C} \cup \bar{\mathcal{C}}$ *is consistent, then* $\Gamma, pc, H \vdash [\bar{\texttt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}]\epsilon : [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}]\tau \setminus [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t}_m \mapsto \bar{\tau}]\mathcal{C} \cup \bar{\mathcal{C}}$*.*

Proof.

By induction on $\Gamma[\bar{\texttt{x}} : \bar{t}], s_p, \emptyset \vdash \epsilon : \tau \setminus \mathcal{C}$. We present only a few cases. The remainder follow in a similar fashion.

Case $\epsilon = \texttt{x}$. Suppose $t = \langle s, f, \alpha \rangle$. Then by (Var), $\Gamma[\bar{\texttt{x}} : \bar{t}], s_p, \emptyset \vdash \epsilon : \langle s \cup s_p, f, \alpha \rangle \setminus \emptyset$.

Since $\texttt{x} \mapsto \llparenthesis v \rrparenthesis^{S_v}$, we have two cases.

Subcase v is a constant. Then by (Val), $\Gamma, pc, H \vdash \llparenthesis v \rrparenthesis^{S_v} : \langle S_v \cup pc, \emptyset, \texttt{int} \rangle \setminus \emptyset$.

Hence $\Gamma, pc, H \vdash [\llparenthesis v \rrparenthesis^{S_v} \mapsto \texttt{x}]\texttt{x} : [s_p \mapsto pc, s \mapsto S_v \cup pc, f \mapsto \emptyset, \alpha \mapsto \texttt{int}]\langle s \cup s_p, f, \alpha \rangle \setminus \emptyset$, and by (Sub), $\Gamma, pc, H \vdash [\llparenthesis v \rrparenthesis^{S_v} \mapsto \texttt{x}]\texttt{x} : [s_p \mapsto pc, s \mapsto S_v \cup pc, f \mapsto \emptyset, \alpha \mapsto \texttt{int}]\langle s \cup s_p, f, \alpha \rangle \setminus \bar{\mathcal{C}}$.

(Field-RC)	$\llparenthesis \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}.\mathtt{f} \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}.\mathtt{f} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
(New-RC)	$\llparenthesis \mathtt{new}\ \mathtt{C}(\bar{\mathcal{V}}, \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
	$\leadsto \llparenthesis \mathtt{new}\ \mathtt{C}(\bar{\mathcal{V}}, \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Invk-RC)	$\llparenthesis \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}.\mathtt{m}(\llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}.\mathtt{m}(\llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
(IArg-RC)	$\llparenthesis \llparenthesis o \rrparenthesis^{S_v}_{I_v}.\mathtt{m}(\bar{\mathcal{V}}, \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
	$\leadsto \llparenthesis \llparenthesis o \rrparenthesis^{S_v}_{I_v}.\mathtt{m}(\bar{\mathcal{V}}, \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Super-RC)	$\llparenthesis \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}.\mathtt{super}(\llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
	$\leadsto \llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}.\mathtt{super}(\llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(SArg-RC)	$\llparenthesis \llparenthesis o \rrparenthesis^{S_v}_{I_v}.\mathtt{super}(\bar{\mathcal{V}}, \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
	$\leadsto \llparenthesis \llparenthesis o \rrparenthesis^{S_v}_{I_v}.\mathtt{super}(\bar{\mathcal{V}}, \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(If-RC)	$\llparenthesis \mathtt{if}\ \llparenthesis \mathcal{E} \rrparenthesis^{S_\mathsf{e}}_{I_\mathsf{e}}\ \mathtt{then}\ \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}\ \mathtt{else}\ \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \rrparenthesis^{S}_{I}, H$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
	$\leadsto \llparenthesis \mathtt{if}\ \llparenthesis \mathcal{E}' \rrparenthesis^{S'_\mathsf{e}}_{I'_\mathsf{e}}\ \mathtt{then}\ \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}\ \mathtt{else}\ \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Seq-RC)	$\llparenthesis \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}; \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}} \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}; \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}}_{\bar{I}} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
(Op-RC)	$\llparenthesis \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1} \oplus \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}, H', \iota', \omega'$
	$\leadsto \llparenthesis \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1} \oplus \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Op-RC')	$\llparenthesis \mathcal{V} \oplus \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e} \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis \mathcal{V} \oplus \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
(Cast-RC)	$\llparenthesis (\mathtt{C})\ \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e} \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis (\mathtt{C})\ \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
(Input-RC)	$\llparenthesis \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}, H', \iota', \omega'$
	$\leadsto \llparenthesis \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Output-RC)	$\llparenthesis \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}, \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}, H', \iota', \omega'$
	$\leadsto \llparenthesis \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}, \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Output-RC')	$\llparenthesis \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\mathcal{V}, \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}, H', \iota', \omega'$
	$\leadsto \llparenthesis \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\mathcal{V}, \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Decl-RC)	$\llparenthesis \mathtt{Declassify}(\llparenthesis \mathcal{E} \rrparenthesis^{S_\mathsf{e}}_{I_\mathsf{e}}, \mathtt{L}) \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
	$\leadsto \llparenthesis \mathtt{Declassify}(\llparenthesis \mathcal{E}' \rrparenthesis^{S'_\mathsf{e}}_{I'_\mathsf{e}}, \mathtt{L}) \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Assign-RC)	$\llparenthesis \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}.\mathtt{f} := \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \rrparenthesis^{S}_{I}, H, \iota, \omega$	if $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}_{I_1}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}, H', \iota', \omega'$
	$\leadsto \llparenthesis \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}_{I'_1}.\mathtt{f} := \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}_{I_2} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	
(Assign-RC')	$\llparenthesis \mathcal{V}.\mathtt{f} := \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e} \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis \mathcal{V}.\mathtt{f} := \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
(Return-RC)	$\llparenthesis \mathtt{return}\ \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e} \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis \mathtt{return}\ \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$
(Val-RC)	$\llparenthesis \llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e} \rrparenthesis^{S}_{I}, H, \iota, \omega \leadsto \llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e} \rrparenthesis^{S}_{I}, H', \iota', \omega'$	if $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}_{I_e}, H, \iota, \omega \leadsto \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}_{I'_e}, H', \iota', \omega'$

Figure 10: Operational Semantics Reductions Under Context

Subcase v is an object identifier.

Then by (Heap), $\Gamma, pc, H \vdash \llparenthesis v \rrparenthesis^{S_v} : \langle pc \cup S_v \cup S'_v, \{\bar{\mathtt{f}} = \bar{t'}\}, \mathtt{C} \rangle \backslash \{\bar{t'} <: \textbf{set}\ \bar{\tau}\}$. Hence $\Gamma, pc, H \vdash [\llparenthesis v \rrparenthesis^{S_v} \mapsto \mathtt{x}]\mathtt{x} : [s_p \mapsto pc, s \mapsto pc \cup S_v \cup S'_v, f \mapsto \{\bar{\mathtt{f}} = \bar{t}\}, \alpha \mapsto \mathtt{C}]\langle s \cup s_p, f, \alpha \rangle \backslash \{\bar{t'} <: \textbf{set}\ \bar{\tau}\}$, and since $\{\bar{t'} <: \textbf{set}\ \bar{\tau}\} \subseteq \bar{\mathcal{C}}$, by (Sub), $\Gamma, pc, H \vdash [\llparenthesis v \rrparenthesis^{S_v} \mapsto \mathtt{x}]\mathtt{x} : [s_p \mapsto pc, s \mapsto pc \cup S_v \cup S'_v, f \mapsto \{\bar{\mathtt{f}} = \bar{t}\}, \alpha \mapsto \mathtt{C}]\langle s \cup s_p, f, \alpha \rangle \backslash \bar{\mathcal{C}}$.

Case $\epsilon = \mathtt{e}'.\mathtt{f}$.

By (Field), if $\Gamma[\bar{\mathtt{x}} : \bar{t}], s_p, \emptyset \vdash \mathtt{e}' : \langle \mathcal{S}', \mathcal{F}', \mathcal{A}' \rangle \backslash \mathcal{C}'$, $\Gamma[\bar{\mathtt{x}} : \bar{t}], s_p, \emptyset \vdash \mathtt{e}'.\mathtt{f} : \langle s_l, f_l, \alpha_l \rangle \backslash \mathcal{C}' \cup \{\langle \mathcal{S}' \cup \mathcal{F}'.\mathtt{f}.\mathsf{S}, \mathcal{F}'.\mathtt{f}.fF, \mathcal{F}'.\mathtt{f}.\mathsf{S} \rangle <: \textbf{get}\ \langle s_l, f_l, \alpha_l \rangle\}$

By induction, $\Gamma, pc, H \vdash [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S}_v}]\mathtt{e}' : [\bar{t}_l \mapsto \bar{t}'_l][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}]\langle \mathcal{S}', \mathcal{F}', \mathcal{A}' \rangle \backslash [\bar{t}_l \mapsto \bar{t}'_l][s_p \mapsto pc, \bar{t}_m \mapsto \bar{\tau}]\mathcal{C}' \cup \bar{\mathcal{C}}$.

So, by (Field), $\Gamma, pc, H \vdash [\bar{\mathtt{x}} \mapsto \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}]\mathtt{e}'.\mathtt{f} : [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}]\langle\, s', f', \alpha' \,\rangle \backslash [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t}_m \mapsto \bar{\tau}]\mathcal{C}' \cup \bar{\mathcal{C}} \cup \{[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t}_m \mapsto \bar{\tau}]\langle\, \mathcal{S}' \cup \mathcal{F}'.\mathtt{f}.\mathsf{S}, \mathcal{F}'.\mathtt{f}.fF, \mathcal{F}'.\mathtt{f}.\mathsf{S} \,\rangle <: \textbf{get}\,\langle\, s_l', f_l', \alpha_l' \,\rangle\}$, which is $\Gamma, pc, H \vdash [\bar{\mathtt{x}} \mapsto \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}]\mathtt{e}'.\mathtt{f} : [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}]\langle\, s', f', \alpha' \,\rangle \backslash [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t}_m \mapsto \bar{\tau}](\mathcal{C}' \cup \{\langle\, \mathcal{S}' \cup \mathcal{F}'.\mathtt{f}.\mathsf{S}, \mathcal{F}'.\mathtt{f}.fF, \mathcal{F}'.\mathtt{f}.\mathsf{S} \,\rangle <: \textbf{get}\,\langle\, s_l, f_l, \alpha_l \,\rangle\}) \cup \bar{\mathcal{C}}$.

□

Lemma 4.8 (Method Substitution) *If $\Gamma, H, pc, \mathcal{C} \vdash_{con} \llparenthesis\, o \,\rrparenthesis^{S_v}.\mathtt{m}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}) : S$, and $mbody(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{s}}$, then there exists a typing $\Gamma, H, pc, \mathcal{C} \vdash_{con} [\bar{\mathtt{x}} \mapsto \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis\, o \,\rrparenthesis^{S_v}]\bar{\mathtt{s}} : S'$, such that $S' \subseteq S$.*

Proof.

By (Invoke) $\Gamma, pc, H \vdash \llparenthesis\, o \,\rrparenthesis^{S_v}.\mathtt{m}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}) : \langle\, \mathcal{S} \cup s, f, \alpha \,\rangle \backslash \mathcal{C}_o \cup \bar{\mathcal{C}} \cup \{\mathtt{C}.\mathtt{m}(\bar{\tau}, \tau \xrightarrow{pc \cup \mathcal{S}} t_r')\}$. By Method Typing, $\Gamma[\bar{\mathtt{x}} \mapsto \bar{t}, \mathtt{this} \mapsto t_t], s_p, H \vdash \bar{\mathtt{s}} : \tau_m \backslash \mathcal{C}_m \cup \{\tau_m <: t_r\}$. By (Invoke) and closure rule $(Method)$, $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau, t_r \mapsto t_r'](\mathcal{C}_m \cup \{\tau_m <: t_r\})$ is consistent. So by Substitution Lemma 4.7, $\Gamma, pc, H \vdash [\bar{\mathtt{x}} \mapsto \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis\, o \,\rrparenthesis^{S_v}]\bar{\mathtt{s}} : [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau, t_r \mapsto t_r']\tau_m \backslash [\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t}_m \mapsto \bar{\tau}, t_t \mapsto \tau, t_r \mapsto t_r'](\mathcal{C}_m \cup \{\tau_m <: t_r\}) \cup \mathcal{C}_o \cup \bar{\mathcal{C}}$.

Hence $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau, t_r \mapsto t_r']\tau_m <: t_r \in \mathcal{C}$, since $\mathcal{C}$ is the closed constraint set from the top-level typing. Applying the substitution $t_r \mapsto t_r'$, we have $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau]\tau_m <: t_r' \in \mathcal{C}$.

Let $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau]\tau_m = \langle\, \mathcal{S}_m, \mathcal{F}_m, \mathcal{A}_m \,\rangle$ and $t_r' = \langle\, s, f, \alpha \,\rangle$. Let $\mathtt{l}$ be any concrete label in S'. We have three cases according to Definition 4.1.

Case $\mathtt{l} \in \mathcal{S}_m$. Since $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau]\tau_m <: t_r' \in \mathcal{C}$, by ($\mathcal{S}$-Union), we have $\mathtt{l} <: s$. So, by Definition 4.1, $\mathtt{l} \in S$.

Case There exists an $s_m \in \mathcal{S}_m$, such that $\mathtt{l} <: s_m \in \mathit{Closure}(LT, \mathcal{C})$. Since $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau]\tau_m <: t_r' \in \mathcal{C}$, by ($\mathcal{S}$-Union), we have $s_m <: s$. So, by ($\mathcal{S}$-Trans) and Definition 4.1, $\mathtt{l} \in S$.

Case There exists an $f_m.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathcal{S}$, such that $\mathtt{l} <: f_m.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathit{Closure}(LT, \mathcal{C})$.

Since $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto pc, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau]\tau_m <: t_r' \in \mathcal{C}$, by ($\mathcal{S}$-Union), we have $f_m.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} <: s$. So, by ($\mathcal{S}$-Trans) and Definition 4.1, $\mathtt{l} \in S$.

So, for any $\mathtt{l} \in S'$, we have $\mathtt{l} \in S$, hence $S' \subseteq S$.

□

Lemma 4.9 (Fixed Point) *For some ι, ω, if $\llbracket\, \Gamma, H_i, S_p, \mathcal{C}, \mathtt{e} \,\rrbracket = \llparenthesis\, \mathcal{E} \,\rrparenthesis^{S}, H$ and $\llparenthesis\, \mathcal{E} \,\rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}, H', \iota', \omega'$, and $\forall o \in dom(H), H(o) = \llparenthesis\, \mathtt{new}\ \mathtt{C}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S}}) \,\rrparenthesis^{S_o}$ then there exists a derivation $\llparenthesis\, \mathcal{E} \,\rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow \llparenthesis\, \mathcal{E}'' \,\rrparenthesis^{S''}, H'', \iota', \omega'$ such that $H''(o) = \llparenthesis\, \mathtt{new}\ \mathtt{C}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S}'}) \,\rrparenthesis^{S_o'}$, $S'' = S, S_o' = S_o, \bar{S}' = \bar{S}$.*

Proof.

By induction on the reduction derivation of $\mathcal{E}$.

By Lemma 4.5, S_p is a subset of all secrecy types. Hence it must be a subset of S in translated values $\llparenthesis\, v \,\rrparenthesis^{S}$. Thus, we omit consideration of S_p in this proof, as it always implicitly occurs in the values of each reduction. We assume, without loss of generality, that the typing of $\Gamma, S_p, H \vdash \mathtt{e} : \tau \backslash \mathcal{C}$ does not end in an instance of (Sub). This means each typing ends with a syntax-directed type rule. (If the typing did end in (Sub), the same reasoning would apply, only adding an additional use of (Sub) to end of the derivation.) We present most of the cases, and the remaining cases follow in a similar fashion.

Case (Op-R) $\quad \llparenthesis\, \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c} \oplus \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'} \,\rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow \llparenthesis\, v \,\rrparenthesis^{S_c \cup S_c' \cup S}, H, \iota, \omega$

By type rule (Val), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c} : S_c$ and $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'} : S_c'$. By type rules (Sub) and (Op), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c} \oplus \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'} : S$, such that $S_c \cup S_c' \subseteq S$.

Now, by Definition 4.4, we have $\llbracket\, \Gamma, H_i, S_p, \mathcal{C}, \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c} \oplus \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'} \,\rrbracket = \llparenthesis\, \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c} \oplus \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'} \,\rrparenthesis^{S}, H = \llparenthesis\, \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c} \oplus \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'} \,\rrparenthesis^{S_c \cup S_c' \cup S}, H$, and the lemma follows.

Case (Field-R) $\quad \llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S'}.\mathtt{f}_i \,\rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow \llparenthesis\, v \,\rrparenthesis^{S_i \cup S \cup S'}, H, \iota, \omega$

By premise to (Field-R), we have $H(o) = \llparenthesis\, \mathtt{new}\ \mathtt{C}(\bar{\mathcal{V}}) \,\rrparenthesis^{S_v}$ and $\mathcal{V}_i = \llparenthesis\, v \,\rrparenthesis^{S_i}$

By type rule (Heap), we have $\Gamma, S_p, H \vdash \llparenthesis\, o \,\rrparenthesis^{S'} : \langle\, S' \cup S_v, \{\bar{\mathtt{f}} : \bar{t}\}, \mathtt{C} \,\rangle \backslash \{\langle\, \bar{S}, \bar{\mathcal{F}}, \bar{\mathcal{A}} \,\rangle <: \textbf{set}\, \bar{t}\}$. By (Heap), $\Gamma(o) = \bar{t}$, so by Definition 4.2, $S_i <: t_i$ are all the constraints from concrete labels flowing into t_i in $\mathcal{C}$. So by type rule (Field), closure rule

(Set) and Definition 4.1, we obtain $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S'}.\mathtt{f}_i : S_i \cup S' \cup S_v$. Hence by (Sub), $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S'}.\mathtt{f}_i : S$, such that $S_i \cup S' \cup S_v \subseteq S$.
Now, by Definition 4.4, we have $[\![\Gamma, H_i, S_p, \mathcal{C}, (\!| o |\!)^{S'}.\mathtt{f}_i]\!] = (\!| (\!| o |\!)^{S'}.\mathtt{f}_i |\!)^S, H = (\!| (\!| o |\!)^{S'}.\mathtt{f}_i |\!)^{S_i \cup S \cup S'}, H$, and the lemma follows.

Case (New-R) $(\!| \mathtt{new\ C}((\!| \bar{v} |\!)^{\bar{S_v}})^{\bar{S}} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| (\!| \mathcal{E}' |\!)^{S'}; (\!| \mathtt{return}\ (\!| o |\!)^S |\!)^S; |\!)^S, H[o \mapsto (\!| \mathtt{new\ C}(\overline{(\!| \mathtt{null} |\!)^{\bar{S}}}) |\!)^S], \iota, \omega$
The heaps are the same apart from the addition to the heap on the right side of the reduction. So, for all heap objects they have in common, $S'_o = S_o$ and $\bar{S}'_o = \bar{S}_o$. Since both configurations are labeled with S, the lemma follows.

Case (Invoke-R) $(\!| (\!| o |\!)^{S_v}.\mathtt{m}((\!| \bar{v} |\!)^{\bar{S_v}}) |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathcal{E}' |\!)^{S' \cup S}, H, \iota, \omega$
By assumption, $(\!| (\!| o |\!)^{S_v}.\mathtt{m}((\!| \bar{v} |\!)^{\bar{S_v}}) |\!)^S, H = [\![\Gamma, H_i, S_p, \mathcal{C}, \mathtt{e}]\!]$, so by Definition 4.4, $\mathtt{e} = (\!| o |\!)^{S_v}.\mathtt{m}((\!| \bar{v} |\!)^{\bar{S_v}})$ and $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S_v}.\mathtt{m}((\!| \bar{v} |\!)^{\bar{S_v}}) : S$.
So, by (Invoke), Definition 4.1, and Definition 4.4, $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S_v} : S_v$. By Lemma 4.5[2], $S_p \subseteq S_v$.
By premise to (Invoke-R), $(\!| \mathcal{E}' |\!)^{S'} = [\![\emptyset, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto (\!| \bar{v} |\!)^{\bar{S_v}}, \mathtt{this} \mapsto (\!| o |\!)^{S_v}]\bar{\mathtt{s}}]\!]$, so by Definition 4.4 $\Gamma, H, S_v, \mathcal{C} \vdash_{con} [\bar{\mathtt{x}} \mapsto (\!| \bar{v} |\!)^{\bar{S_v}}, \mathtt{this} \mapsto (\!| o |\!)^{S_v}]\bar{\mathtt{s}} : S'$. Since $S_p \subseteq S_v$, by Lemma 4.6[2], $\Gamma, H, S_p, \mathcal{C} \vdash_{con} [\bar{\mathtt{x}} \mapsto (\!| \bar{v} |\!)^{\bar{S_v}}, \mathtt{this} \mapsto (\!| o |\!)^{S_v}]\bar{\mathtt{s}} : S'$.
Since $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S_v}.\mathtt{m}((\!| \bar{v} |\!)^{\bar{S_v}}) : S$, by Concrete Substitution Lemma 4.8, $S' \subseteq S$. The lemma follows by (Sub) and Definition 4.4.

Case (Super-R) follows in a similar manner to (Invoke-R).

Case (Super-R') follows in a similar manner to (Invoke-R).

Case (Assign-R) $(\!| (\!| o |\!)^{S'}.\mathtt{f} := (\!| v |\!)^{S_v}; |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathtt{null} |\!)^{S \cup S_v \cup S'}, H[o \mapsto (\!| \mathtt{new\ C}(\ldots, (\!| v |\!)^{S_v \cup S \cup S' \cup S_i}, \ldots) |\!)^{S''}], \iota, \omega$
By premise to (Assign-R), we have $H(o) = (\!| \mathtt{new\ C}(\bar{\mathcal{V}}) |\!)^{S''}$. By type rule (Heap), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S'} : S' \cup S''$, and by type rule (Val), $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| v |\!)^{S_v} : S_v$. So, by type rule (F-Assign), $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S'}.\mathtt{f} := (\!| v |\!)^{S_v};: S' \cup S'' \cup S_v$ and by (Sub), $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| o |\!)^{S'}.\mathtt{f} := (\!| v |\!)^{S_v};: S$, where $S' \cup S'' \cup S_v \subseteq S$.
By Definition 4.4, we have $[\![\Gamma, H_i, S_p, \mathcal{C}, (\!| o |\!)^{S'}.\mathtt{f} := (\!| v |\!)^{S_v};]\!] = (\!| (\!| o |\!)^{S'}.\mathtt{f} := (\!| v |\!)^{S_v}; |\!)^S, H = (\!| (\!| o |\!)^{S'}.\mathtt{f} := (\!| v |\!)^{S_v}; |\!)^{S \cup S_v \cup S'}, H$.
Now, by type rule (F-Assign), we have the constraint $\langle s_i, f_i, \alpha_i \rangle <:$ **set** $\langle S' \cup S'' \cup S_v, \mathcal{F}, \mathcal{A} \rangle \in \mathcal{C}$, for $\Gamma(o) = \overline{\langle s, f, \alpha \rangle}$.
Thus, $S' \cup S'' \cup S_v <: s_i \in \mathcal{C}$. According to Definition 4.2, $S' \cup S'' \cup S_v <: s_i \in \mathcal{C}$ means $S' \cup S'' \cup S_v$ was placed on the field in the heap. Thus, $S' \cup S'' \cup S_v \subseteq S_i$. Since $S' \cup S'' \cup S_v \subseteq S$, we have $S_v \cup S \cup S' \cup S_i = S_i$. The lemma follows, since for the only change on the heap, $S_v \cup S \cup S' \cup S_i = S_i$.

Case (IfTrue-R) $(\!| \mathtt{if}\ (\!| \mathtt{True} |\!)^{S_v} \mathtt{then}\ (\!| \mathcal{E}_1 |\!)^{S_1}\ \mathtt{else}\ (\!| \mathcal{E}_2 |\!)^{S_2} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathcal{E}_1 |\!)^{S_1 \cup S}, H, \iota, \omega$
Since $[\![\Gamma, H_i, S_p, \mathcal{C}, \mathtt{e}]\!] = (\!| \mathtt{if}\ (\!| \mathtt{True} |\!)^{S_v} \mathtt{then}\ (\!| \mathcal{E}_1 |\!)^{S_1}\ \mathtt{else}\ (\!| \mathcal{E}_2 |\!)^{S_2} |\!)^S, H$, there exists $\bar{\mathtt{s}}_1$ and $\bar{\mathtt{s}}_2$ such that
$\mathtt{e} = \mathtt{if}\ (\!| \mathtt{True} |\!)^{S_v}\ \mathtt{then}\ \{\bar{\mathtt{s}}_1\}\ \mathtt{else}\ \{\bar{\mathtt{s}}_2\}$. By Definition 4.4, we know $[\![\Gamma, H_i, S_p \cup S_v, \mathcal{C}, \{\bar{\mathtt{s}}_1\}]\!] = (\!| \mathcal{E}_1 |\!)^{S_1}, H$, and $[\![\Gamma, H_i, S_p \cup S_v, \mathcal{C}, \{\bar{\mathtt{s}}_2\}]\!] = (\!| \mathcal{E}_2 |\!)^{S_2}, H$, and $[\![\Gamma, H_i, S_p, \mathcal{C}, (\!| \mathtt{True} |\!)^{S_v}]\!] = (\!| \mathtt{True} |\!)^{S_v}, H$.
Thus, $\Gamma, H, S_p \cup S_v, \mathcal{C} \vdash_{con} \{\bar{\mathtt{s}}_1\} : S_1$, and $\Gamma, H, S_p \cup S_v, \mathcal{C} \vdash_{con} \{\bar{\mathtt{s}}_2\} : S_2$, and $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| \mathtt{True} |\!)^{S_v} : S_v$.
Now, by typerule (If) and Definition 4.1, $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \mathtt{if}\ (\!| \mathtt{True} |\!)^{S_v} \mathtt{then}\ (\!| \mathcal{E}_1 |\!)^{S_1}\ \mathtt{else}\ (\!| \mathcal{E}_2 |\!)^{S_2} : S_v \cup S_1 \cup S_2$. So, we have $S_v \cup S_1 \cup S_2 \subseteq S$, and the lemma follows.

Case (Input-R) $(\!| \mathtt{read_L}((\!| \mathtt{fd} |\!)^{S_f}) |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathtt{c} |\!)^{\mathtt{L}}, H, \iota', \omega$
By type rule (Input), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \mathtt{read_L}((\!| \mathtt{fd} |\!)^{S_f}) : S_f \cup \mathtt{L}$. By (Sub), $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \mathtt{read_L}((\!| \mathtt{fd} |\!)^{S_f}) : S$, where $S_f \cup \mathtt{L} \subseteq S$.
By Definition 4.4, $[\![\Gamma, H_i, S_p, \mathcal{C}, \mathtt{read_L}((\!| \mathtt{fd} |\!)^{S_f})]\!] = (\!| \mathtt{read_L}((\!| \mathtt{fd} |\!)^{S_f}) |\!)^S, H$. Since by (Input-R), $S \cup S_f \subseteq \mathtt{L}$, the lemma follows.

Case (Output-R) $(\!| \mathtt{write_L}((\!| \mathtt{c} |\!)^{S_c}, (\!| \mathtt{fd} |\!)^{S_f}) |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathtt{null} |\!)^{S_c \cup S_f \cup S}, H, \iota, \omega'$
By type rule (Val), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| \mathtt{c} |\!)^{S_c} : S_c$ and $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| \mathtt{c} |\!)^{S_f} : S_f$. By type rule (Output), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \mathtt{write_L}((\!| \mathtt{c} |\!)^{S_c}, (\!| \mathtt{fd} |\!)^{S_f}) : S_c \cup S_f$. By (Sub), $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \mathtt{write_L}((\!| \mathtt{c} |\!)^{S_c}, (\!| \mathtt{fd} |\!)^{S_f}) : S$, where $S_c \cup S_f \subseteq S$. Now, by Definition 4.4, we have $[\![\Gamma, H_i, S_p, \mathcal{C}, \mathtt{write_L}((\!| \mathtt{c} |\!)^{S_c}, (\!| \mathtt{fd} |\!)^{S_f})]\!] = (\!| \mathtt{write_L}((\!| \mathtt{c} |\!)^{S_c}, (\!| \mathtt{fd} |\!)^{S_f}) |\!)^S, H = (\!| \mathtt{write_L}((\!| \mathtt{c} |\!)^{S_c}, (\!| \mathtt{fd} |\!)^{S_f}) |\!)^{S_c \cup S_f \cup S}, H$, and the lemma follows.

Case (Declassify-R) $(\!| \mathtt{Declassify}((\!| v |\!)^{S_v}, \mathtt{L}) |\!)^S, H, \iota, \omega \rightsquigarrow (\!| v |\!)^{(S_v - \mathtt{L}) \cup S}, H, \iota, \omega$
By type rule (Val) or (Heap), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} (\!| v |\!)^{S_v} : S'$, where $S_v \subseteq S'$. By type rule (Declassify), we have $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \mathtt{Declassify}((\!| v |\!)^{S'}, \mathtt{L}) : S' - \mathtt{L}$. By (Sub), $\Gamma, H, S_p, \mathcal{C} \vdash_{con} \mathtt{Declassify}((\!| v |\!)^{S'}, \mathtt{L}) : S$, where $S' - \mathtt{L} \subseteq S$. Now, by Definition 4.4, we have $[\![\Gamma, H_i, S_p, \mathcal{C}, \mathtt{Declassify}((\!| v |\!)^{S_v}, \mathtt{L})]\!] = (\!| \mathtt{Declassify}((\!| v |\!)^{S_v}, \mathtt{L}) |\!)^S, H = (\!| \mathtt{Declassify}((\!| v |\!)^{S_v}, \mathtt{L}) |\!)^{(S_v - \mathtt{L}) \cup S}, H$, and the lemma follows.

Case (Field-RC) $(\!| (\!| \mathcal{E} |\!)^{S_e}.\mathtt{f} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| (\!| \mathcal{E}' |\!)^{S'_e}.\mathtt{f} |\!)^S, H', \iota', \omega'$

By induction on the reduction $(\!| \mathcal{E} |\!)^{S_e}, H, \iota, \omega \rightsquigarrow (\!| \mathcal{E}' |\!)^{S'_e}, H', \iota', \omega'$, we have $S'_o = S_o, \bar{S}'_o = \bar{S}_o$. Since the outer label, S is unchanged in the reduction, the lemma follows.

□

Lemma 4.10 shows that a reduction of a translated expression produces an configuration that is also a translated expression. This is needed in the proofs of soundness and noninterference, that continually use Fixed Point Lemma 4.9, which requires a translated configuration as a pre-condition.

Lemma 4.10 (Uniformity of Translated Expressions) *For some ι, ω, if $[\![\Gamma, H_i, pc, \mathcal{C}, \mathtt{e}]\!] = (\!| \mathcal{E} |\!)^S, H$ and $(\!| \mathcal{E} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathcal{E}' |\!)^{S'}, H', \iota', \omega'$, then there exists an expression $\mathtt{e}'$, a heap H'_i, a type environment Γ, and a pc', such that $[\![\Gamma', H'_i, pc', \mathcal{C}, \mathtt{e}']\!] = (\!| \mathcal{E}' |\!)^{S'}, H'$.*

Proof.

By induction on the reduction derivation of $\mathcal{E}$.

We assume, without loss of generality, that the typing of $\Gamma, pc, H \vdash \mathtt{e} : \tau \backslash \mathcal{C}$ does not end in an instance of (Sub). This means each typing ends with a syntax-directed type rule. (If the typing did end in (Sub), the same reasoning would apply, only adding an additional use of (Sub) to end of the derivation.)

Case (Op-R) $\quad (\!| (\!| \mathtt{c} |\!)^{S_c} \oplus (\!| \mathtt{c}' |\!)^{S'_c} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| v |\!)^{S_c \cup S'_c \cup S}, H, \iota, \omega$

By type rule (Val), we have $\Gamma, H, pc, \mathcal{C} \vdash_{con} (\!| v |\!)^{S_c \cup S'_c \cup S} : S_c \cup S'_c \cup S$. Then, by Definition 4.4, we have $[\![\Gamma, H_i, pc, \mathcal{C}, (\!| v |\!)^{S_c \cup S'_c \cup S}]\!] = (\!| v |\!)^{S_c \cup S'_c \cup S}, H$. The lemma follows.

Cases (Field-R), (Cast-R), (Input-R), (Output-R), (Declassify-R), (Return-R), (Skip-R), (Super-R'), and (SubVal-R) all follow in a similar manner to (Op-R), since they all reduce to values.

Case (New-R) $\quad (\!| \mathtt{new\ C}((\!| \bar{v} |\!)^{\bar{S}_v})^{\bar{S}} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| (\!| \mathcal{E}' |\!)^{S'}; (\!| \mathtt{return}\ (\!| o |\!)^S |\!)^S; |\!)^S, H[o \mapsto (\!| \mathtt{new\ C}((\!| \overline{\mathtt{null}} |\!)^{\bar{S}}) |\!)^S], \iota, \omega$

By premise to (New-R), we have $cnbody(\mathtt{C}) = \mathtt{super}(\bar{\mathtt{e}}); \bar{\mathtt{s}}$ and $(\!| \mathcal{E}' |\!)^{S'} = [\![\Gamma', H, S, \mathcal{C}, [\bar{\mathtt{x}} \mapsto (\!| \bar{v} |\!)^{\bar{S}_v}, \mathtt{this} \mapsto (\!| o |\!)^S]\mathtt{this.super}(\mathtt{D}, \bar{\mathtt{e}}); \bar{\mathtt{s}}]\!]_e$, where $\Gamma' = \Gamma[o \mapsto \bar{t}]$. By Definition 4.3, (Seq), (Return), and (Heap), we have $[\![\Gamma', H, S, \mathcal{C}, [\bar{\mathtt{x}} \mapsto (\!| \bar{v} |\!)^{\bar{S}}, \mathtt{this} \mapsto (\!| o |\!)^S]\mathtt{this.super}(\mathtt{D}, \bar{\mathtt{e}}); \bar{\mathtt{s}}; \mathtt{return}\ (\!| o |\!)^S;]\!]_e = (\!| (\!| \mathcal{E}' |\!)^{S'}; (\!| \mathtt{return}\ (\!| o |\!)^S |\!)^S; |\!)^S$.

We now show the heap translation condition holds. By assumption, Definition 4.3, and (New), we have $\Gamma, pc, H_i \vdash \mathtt{new\ C}((\!| \bar{v} |\!)^{\bar{S}_v}) : \langle \mathcal{S}, \{\bar{\mathtt{f}} : \bar{t}\}, \mathcal{A} \rangle \backslash \mathcal{C}'$ and $\bar{S}' <: \bar{s} \in \mathcal{C}$. Hence, by Definition 4.2, $[\![\Gamma, H_i[o \mapsto (\!| \mathtt{new\ C}((\!| \overline{\mathtt{null}} |\!)^{\bar{S}}) |\!)^S], \mathcal{C}]\!] = H[o \mapsto (\!| \mathtt{new\ C}((\!| \overline{\mathtt{null}} |\!)^{\bar{S}}) |\!)^S]$.

The lemma follows by Definition 4.4.

Case (Invoke-R) $\quad (\!| (\!| o |\!)^{S_v}.\mathtt{m}((\!| \bar{v} |\!)^{\bar{S}_v}) |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathcal{E}' |\!)^{S' \cup S}, H, \iota, \omega$

By premise to (Invoke-R), we have $mbody(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{s}}$ and $(\!| \mathcal{E}' |\!)^{S'} = [\![\emptyset, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto (\!| \bar{v} |\!)^{\bar{S}_v}, \mathtt{this} \mapsto (\!| o |\!)^{S_v}]\bar{\mathtt{s}}]\!]_e$. By (Sub) and Definition 4.4, $(\!| \mathcal{E}' |\!)^{S' \cup S}, H = [\![\emptyset, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto (\!| \bar{v} |\!)^{\bar{S}_v}, \mathtt{this} \mapsto (\!| o |\!)^{S_v}]\bar{\mathtt{s}}]\!]$

Case (Super-R) follows in a similar manner to (New-R) and (Invoke-R).

Case (IfTrue-R) $\quad (\!| \mathtt{if}\ (\!| \mathtt{True} |\!)^{S_v} \mathtt{then}\ (\!| \mathcal{E}_1 |\!)^{S_1}\ \mathtt{else}\ (\!| \mathcal{E}_2 |\!)^{S_2} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathcal{E}_1 |\!)^{S_1 \cup S}, H, \iota, \omega$

By Definition 4.4, we have $(\!| \mathcal{E}_1 |\!)^{S_1} = [\![\Gamma, H_i, pc \cup S_v, \mathcal{C}, \{\bar{\mathtt{s}}\}]\!]$. By (Sub) and Definition 4.4, $(\!| \mathcal{E}_1 |\!)^{S_1 \cup S} = [\![\Gamma, H_i, pc \cup S_v, \mathcal{C}, \{\bar{\mathtt{s}}\}]\!]$

Case (Assign-R) $\quad (\!| (\!| o |\!)^{S'}.\mathtt{f} := (\!| v |\!)^{S_v}; |\!)^S, H, \iota, \omega \rightsquigarrow (\!| \mathtt{null} |\!)^{S \cup S_v \cup S'}, H[o \mapsto (\!| \mathtt{new\ C}(\ldots, (\!| v |\!)^{S_v \cup S \cup S' \cup S_i}, \ldots) |\!)^{S''}], \iota, \omega$

By Definition 4.4, Definition 4.1, and type rule (Val), we have

$[\![\Gamma, H_i[o \mapsto (\!| \mathtt{new\ C}(\ldots, (\!| v |\!)^{S_v \cup S \cup S' \cup S_i}, \ldots) |\!)^{S''}], pc, \mathcal{C}, (\!| \mathtt{null} |\!)^{S \cup S_v \cup S'}]\!] =$
$(\!| \mathtt{null} |\!)^{S \cup S_v \cup S'}, H[o \mapsto (\!| \mathtt{new\ C}(\ldots, (\!| v |\!)^{S_v \cup S \cup S' \cup S_i}, \ldots) |\!)^{S''}]$.

Case (Seq-R) $\quad (\!| (\!| v |\!)^{S_v}; (\!| \bar{\mathcal{E}} |\!)^{\bar{S}} |\!)^S, H, \iota, \omega \rightsquigarrow (\!| (\!| \bar{\mathcal{E}} |\!)^{\bar{S}} |\!)^S, H, \iota, \omega$

According to Definition 4.4, we have

$(\!| \bar{\mathcal{E}} |\!)^{\bar{S}}, H = [\![\Gamma, H, pc, \mathcal{C}, \mathtt{s}_2]\!]; \ldots; [\![\Gamma, H, pc, \mathcal{C}, \mathtt{s}_\mathtt{n}]\!];$

According to type rule (Seq), $\Gamma, H, pc, \mathcal{C} \vdash_{con} \mathtt{s}_2; \ldots; \mathtt{s}_\mathtt{n} : S$, so we have

$[\![\Gamma, H_i, pc, \mathcal{C}, \mathtt{s}_2; \ldots; \mathtt{s}_\mathtt{n};]\!] = (\!| (\!| \bar{\mathcal{E}} |\!)^{\bar{S}} |\!)^S, H$, and the lemma follows.

Case (Block-R) $\llparenthesis \{ \llparenthesis \mathcal{E} \rrparenthesis^{S'} \} \rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E} \rrparenthesis^{S \cup S'}, H, \iota, \omega$

By Definition 4.4, we have $\llparenthesis \mathcal{E} \rrparenthesis^{S'}, H = [\![\Gamma, H_i, pc, \mathcal{C}, \bar{\mathtt{s}}]\!]$.

By premise to type rule (Block), we have $\Gamma, H, pc, \mathcal{C} \vdash_{con} \mathtt{s_2}; \ldots; \mathtt{s_n}; : S$ (in other words, $S = S'$).

So, we have $[\![\Gamma, H_i, pc, \mathcal{C}, \bar{\mathtt{s}}]\!] = \llparenthesis \mathcal{E} \rrparenthesis^{S \cup S'}, H$, and the lemma follows.

Case (Field-RC) $\llparenthesis \llparenthesis \mathcal{E} \rrparenthesis^{S_e}.\mathtt{f} \rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow \llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}.\mathtt{f} \rrparenthesis^{S}, H', \iota', \omega'$

By premise to (Field-RC), we have $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}, H, \iota, \omega \rightsquigarrow \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}, H', \iota, \omega$. By induction, this means $\llparenthesis \mathcal{E} \rrparenthesis^{S_e}, H = [\![\Gamma, H_i, pc, \mathcal{C}, \mathtt{e}]\!]$ and $\llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}, H' = [\![\Gamma', H'_i, pc', \mathcal{C}, \mathtt{e}']\!]$. Now, by Definition 4.4, we have $\Gamma', H', pc', \mathcal{C} \vdash_{con} \mathtt{e}' : S'_e$. According to Fixed Point Lemma 4.9, $S_e = S'_e$, so by type rule (Field), $\Gamma', H', pc', \mathcal{C} \vdash_{con} \mathtt{e}'.\mathtt{f} : S$. So, by Definition 4.4 $\llparenthesis \llparenthesis \mathcal{E}' \rrparenthesis^{S'_e}.\mathtt{f} \rrparenthesis^{S}, H' = [\![\Gamma', H'_i, pc', \mathcal{C}, \mathtt{e}'.\mathtt{f}]\!]$.

Remaining cases: (*-RC)

□

Fixed Point Lemma 4.9 along with Lemma 4.10 produces the following soundness result.

Theorem 4.11 (Soundness) *If $\epsilon : \tau \backslash \mathcal{C}$ and $Closure(LT, \mathcal{C})$ is consistent, then there exists a translation $[\![\emptyset, \emptyset, \emptyset, \mathcal{C}, \epsilon]\!]$, such that $[\![\emptyset, \emptyset, \emptyset, \mathcal{C}, \epsilon]\!], \iota, \omega \not\rightsquigarrow^{*} CkFail, H', \iota', \omega'$.*

Proof. Suppose $[\![\emptyset, \emptyset, \emptyset, \mathcal{C}, \epsilon, \iota, \omega]\!] \rightsquigarrow^{*} CkFail, \iota', \omega'$. Let $[\![\emptyset, \emptyset, \emptyset, \mathcal{C}, \mathtt{e}]\!], \iota, \omega \rightsquigarrow^{*} \llparenthesis \mathcal{E} \rrparenthesis^{S}, H', \iota', \omega'$, be the sequence of reductions immediately before the check failure occurs. We have three cases.

Case $\llparenthesis \mathcal{E} \rrparenthesis^{S} = \llparenthesis \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathtt{fd} \rrparenthesis^{S_f}) \rrparenthesis^{S}$

By Fixed Point Lemma 4.9 and Lemma 4.10), we know there exists a Γ', H', and pc', such that $\llparenthesis \mathcal{E} \rrparenthesis^{S} = [\![\Gamma', H', pc', \mathcal{C}, \epsilon]\!]$. Hence, by Definition 4.4, Definition 4.1, and type rules (Input), (Val) and (Sub), we have $\Gamma', pc', H' \vdash \llparenthesis \mathtt{fd} \rrparenthesis^{S_f} : \langle \mathcal{S}_f, \emptyset, \mathtt{FD} \rangle \backslash \emptyset$, where $S_f \subseteq \mathcal{S}_f$, and $\Gamma', pc', H' \vdash \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathtt{fd} \rrparenthesis^{S_f}) : \langle \mathcal{S}_f \cup \mathtt{L}, \emptyset, \mathtt{int} \rangle \backslash SC(\mathtt{L}, \mathcal{S}_f)$. By assumption, $SC(\mathtt{L}, \mathcal{S}_f)$ is consistent, so according to Definition 3.2, $\mathcal{S}_f \subseteq \mathtt{L}$.

Now, by Definition 4.1 and Definition 4.4, $[\![\Gamma', H', pc', \mathcal{C}, \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathtt{fd} \rrparenthesis^{S_f})]\!] = \llparenthesis \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathtt{fd} \rrparenthesis^{S_f}) \rrparenthesis^{S}$.

By (InFail-R), $S_f \cup \mathtt{L} \not\subseteq \mathtt{L}$. However, we just showed that $S_f \subseteq \mathtt{L}$, and since $\mathtt{L} \subseteq \mathtt{L}$, we have $S_f \cup \mathtt{L} \subseteq \mathtt{L}$, a contradiction. Hence, this step cannot have occurred.

Case $\llparenthesis \mathcal{E} \rrparenthesis^{S} = \llparenthesis \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis \mathtt{c} \rrparenthesis^{S_c}, \llparenthesis \mathtt{fd} \rrparenthesis^{S_f}) \rrparenthesis^{S}$ follows in a similar manner to the previous case.

Case $CkFail$ is a subconfiguration of $\llparenthesis \mathcal{E} \rrparenthesis^{S}$. By induction on the structure of $\llparenthesis \mathcal{E} \rrparenthesis^{S}$.

□

4.4 Noninterference

As stated previously, our Noninterference Theorem 4.24 states that for a typeable program P, any two runs of the program differing only in high input streams will produce the same low output streams, and that the resulting low input streams are also equivalent. A more detailed summary of the proof technique in this section follows.

We translate the program into configurations via Translation Definition 4.4 these map the label types of each sub-expression onto each sub-configuration. Fixed Point Lemma 4.9 assures the existence of a reduction sequence, such that the labels in the reduction sequence, and the labels on the heap will never increase. Translation Lemma 4.10 shows that a reduction step of any translated expression produces a configuration that is also a translated expression; this allows Fixed Point Lemma 4.9 and other lemmas to be applied, since a translated expression is required by assumption.

Now, let $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}$ be the initial configuration of the translated program. Without loss of generality, the reduction of $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}$ with either set of inputs is broken into reductions of alternating zero or more high steps and one low step. According to the High Reduction Lemma 4.23 and Low Reduction Lemma 4.22, if only the high inputs differ, there exists executions of the configuration $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}$ that must make all of the same low reductions (with possibly differing high values); furthermore, the low portions of the respective heaps must be the same. This means that each low input and output step must be the same for both runs, resulting in equivalent low input and output streams. These existentials are due to the non-determinism in the labeled semantics. We then define an unlabeled semantics, which is deterministic and executes identically to the labeled semantics, only without the labels. We then re-state noninterference for this semantics, where the execution deterministically produces equivalent low input and output streams.

We assume `Low` is the set of security labels the low observer has access to, so any subset of this set is accessible to the low observer. Any security labels outside this subset relation are considered *high* e.g. if `High` $\not\subseteq$ `Low`, then `High` is considered a high label and any data that carries this label is considered high data, which must not be revealed to the low observer.

A low computation step is any step that is not affected by high data, as defined in Definition 4.12. Definition 4.13 defines any steps that are not low as high steps, i.e. any step that is affected by data labeled high. Thus, any reduction step $\rightsquigarrow$ is either a high step, or a low step.

Definition 4.12 (Low step) *A low step, denoted* $(\!|\,\mathcal{E}\,|\!)^S, H, \iota, \omega \rightsquigarrow_l (\!|\,\mathcal{E}'\,|\!)^{S'}, H', \iota', \omega'$ *is a step,* $(\!|\,\mathcal{E}\,|\!)^S, H, \iota, \omega \rightsquigarrow (\!|\,\mathcal{E}'\,|\!)^{S'}, H', \iota', \omega'$ *where the root of the context derivation tree contains only a subset of* `Low` *secrecy labels on the right hand side, i.e.* $(\!|\,\mathcal{E}_r\,|\!)^{S_r}, H, \iota, \omega \rightsquigarrow (\!|\,\mathcal{E}'_r\,|\!)^{S'_r}, H', \iota', \omega'$ *is the root of the tree and* $S'_r \subseteq \texttt{Low}$*, or the root of the context derivation tree is a use of* (Invoke-R), (Super-R), or (Super-R') *where the object reference contains only a subset of* `Low` *secrecy labels, e.g.* $(\!|\,(\!|\,o\,|\!)^{S_v}.\texttt{m}(\bar{\mathcal{V}})\,|\!)^S, H, \iota, \omega \rightsquigarrow \mathcal{E}', H', \iota', \omega'$*, and* $S_v \subseteq \texttt{Low}$*.*

Definition 4.13 (High step) *A high step, denoted* $(\!|\,\mathcal{E}\,|\!)^S, H, \iota, \omega \rightsquigarrow_h (\!|\,\mathcal{E}'\,|\!)^{S'}, H', \iota', \omega'$ *is a step,* $(\!|\,\mathcal{E}\,|\!)^S, H, \iota, \omega \rightsquigarrow (\!|\,\mathcal{E}'\,|\!)^{S'}, H', \iota', \omega'$ *where the root of the context derivation tree contains secrecy labels that are not in* `Low` *on the right hand side, i.e.* $(\!|\,\mathcal{E}_r\,|\!)^{S_r}, H, \iota, \omega \rightsquigarrow (\!|\,\mathcal{E}'_r\,|\!)^{S'_r}, H', \iota', \omega'$ *is the root of the tree and* $S'_r \not\subseteq \texttt{Low}$*, and if the root of the context derivation tree is a use of* (Invoke-R), (Super-R), or (Super-R')*, the object reference must contain secrecy labels that are not in* `Low`*, e.g.* $(\!|\,(\!|\,o\,|\!)^{S_v}.\texttt{m}(\bar{\mathcal{V}})\,|\!)^S, H, \iota, \omega \rightsquigarrow \mathcal{E}', H', \iota', \omega'$*, and* $S_v \not\subseteq \texttt{Low}$*.*

Definition 4.14 (Termination) *A configuration* $(\!|\,\mathcal{E}\,|\!)^S, H, \iota, \omega$ *terminates iff there exists a* $(\!|\,v\,|\!)^{S'}$*,* H'*,* ι'*, and* ω' *such that* $(\!|\,\mathcal{E}\,|\!)^S, H, \iota, \omega \rightsquigarrow^* (\!|\,v\,|\!)^{S'}, H', \iota', \omega'$

Definition 4.15 specifies the bisimulation relation between configurations, heaps, and input and output streams.

Definition 4.15 (Bisimulation Relation)

1. *(Labels).*
   $S_1 \simeq_{\texttt{Low}} S_2$ *iff either*
   (a) $S_1 \subseteq \texttt{Low}$, $S_2 \subseteq \texttt{Low}$*, and* $S_1 = S_2$*; or*
   (b) $S_1 \not\subseteq \texttt{Low}$ *and* $S_2 \not\subseteq \texttt{Low}$.
2. *(Labeled Configurations).* $(\!|\,\mathcal{E}_1\,|\!)^{S_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}_2\,|\!)^{S_2}$ *iff either*
   (a) $\mathcal{E}_1 = v_1$, $\mathcal{E}_2 = v_2$ *and either*
      i. $S_1 \subseteq \texttt{Low}$, $S_2 \subseteq \texttt{Low}$, $S_1 = S_2$*, and* $v_1 = v_2$*; or*
      ii. $S_1 \not\subseteq \texttt{Low}$ *and* $S_2 \not\subseteq \texttt{Low}$*; or*
   (b) $S_1 \simeq_{\texttt{Low}} S_2$ *and* $\mathcal{E}_1 = \mathcal{E}_2 = \texttt{x}$*, for some* `x`*; or*
   (c) $S_1 \simeq_{\texttt{Low}} S_2$ *and* $\mathcal{E}_1 = \mathcal{E}_2 = \texttt{this}$*; or*
   (d) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = (\!|\,\mathcal{E}'_1\,|\!)^{S'_1}.\texttt{f}$, $\mathcal{E}_2 = (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}.\texttt{f}$*, and* $(\!|\,\mathcal{E}'_1\,|\!)^{S'_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}$*; or*
   (e) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = (\texttt{C})\ (\!|\,\mathcal{E}'_1\,|\!)^{S'_1}$, $\mathcal{E}_2 = (\texttt{C})\ (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}$*, and* $(\!|\,\mathcal{E}'_1\,|\!)^{S'_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}$*; or*
   (f) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = (\!|\,\mathcal{E}'_1\,|\!)^{S'_1} \oplus (\!|\,\mathcal{E}''_1\,|\!)^{S''_1}$, $\mathcal{E}_2 = (\!|\,\mathcal{E}'_2\,|\!)^{S'_2} \oplus (\!|\,\mathcal{E}''_2\,|\!)^{S''_2}$*, and* $(\!|\,\mathcal{E}'_1\,|\!)^{S'_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}$, $(\!|\,\mathcal{E}''_1\,|\!)^{S''_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}''_2\,|\!)^{S''_2}$*; or*
   (g) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \texttt{new C}(\overline{(\!|\,\mathcal{E}'_1\,|\!)^{S'_1}})^{\bar{S}''_1}$, $\mathcal{E}_2 = \texttt{new C}(\overline{(\!|\,\mathcal{E}'_2\,|\!)^{S'_2}})^{\bar{S}''_2}$*, and* $\overline{(\!|\,\mathcal{E}'_1\,|\!)^{S'_1}} \simeq_{\texttt{Low}} \overline{(\!|\,\mathcal{E}'_2\,|\!)^{S'_2}}$, $\bar{S}''_1 \simeq_{\texttt{Low}} \bar{S}''_2$*; or*
   (h) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = (\!|\,\mathcal{E}'_1\,|\!)^{S'_1}.\texttt{m}(\overline{(\!|\,\mathcal{E}''_1\,|\!)^{S''_1}})$, $\mathcal{E}_2 = (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}.\texttt{m}(\overline{(\!|\,\mathcal{E}''_2\,|\!)^{S''_2}})$*, and* $(\!|\,\mathcal{E}'_1\,|\!)^{S'_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}$, $\overline{(\!|\,\mathcal{E}''_1\,|\!)^{S''_1}} \simeq_{\texttt{Low}} \overline{(\!|\,\mathcal{E}''_2\,|\!)^{S''_2}}$*; or*
   (i) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \texttt{read}_{(\texttt{L},\texttt{L}')}((\!|\,\mathcal{E}'_1\,|\!)^{S'_1})$, $\mathcal{E}_2 = \texttt{read}_{(\texttt{L},\texttt{L}')}((\!|\,\mathcal{E}'_2\,|\!)^{S'_2})$*, and* $(\!|\,\mathcal{E}'_1\,|\!)^{S'_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}$*; or*
   (j) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \texttt{write}_{(\texttt{L},\texttt{L}')}((\!|\,\mathcal{E}'_1\,|\!)^{S'_1}, (\!|\,\mathcal{E}''_1\,|\!)^{S''_1})$, $\mathcal{E}_2 = \texttt{write}_{(\texttt{L},\texttt{L}')}((\!|\,\mathcal{E}'_2\,|\!)^{S'_2}, (\!|\,\mathcal{E}''_2\,|\!)^{S''_2})$*, and* $(\!|\,\mathcal{E}'_1\,|\!)^{S'_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}'_2\,|\!)^{S'_2}$, $(\!|\,\mathcal{E}''_1\,|\!)^{S''_1} \simeq_{\texttt{Low}} (\!|\,\mathcal{E}''_2\,|\!)^{S''_2}$*; or*

(k) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \texttt{if}\ \llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'}\ \texttt{then}\ \llparenthesis \mathcal{E}_1'' \rrparenthesis^{S_1''}\ \texttt{else}\ \llparenthesis \mathcal{E}_1''' \rrparenthesis^{S_1'''}$, $\mathcal{E}_2 = \texttt{if}\ \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}\ \texttt{then}\ \llparenthesis \mathcal{E}_2'' \rrparenthesis^{S_2''}\ \texttt{else}\ \llparenthesis \mathcal{E}_2''' \rrparenthesis^{S_2'''}$, *and* $\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}$, $\llparenthesis \mathcal{E}_1'' \rrparenthesis^{S_1''} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2'' \rrparenthesis^{S_2''}$, $\llparenthesis \mathcal{E}_1''' \rrparenthesis^{S_1'''} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2''' \rrparenthesis^{S_2'''}$*; or*

(l) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \overline{\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'}}$, $\mathcal{E}_2 = \overline{\llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}}$, *and* $\overline{\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'}} \simeq_{\texttt{Low}} \overline{\llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}}$*; or*

(m) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \{\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'}\}$, $\mathcal{E}_2 = \{\llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}\}$, *and* $\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}$*; or*

(n) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'}.\texttt{f} := \llparenthesis \mathcal{E}_1'' \rrparenthesis^{S_1''};$, $\mathcal{E}_2 = \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}.\texttt{f} := \llparenthesis \mathcal{E}_2'' \rrparenthesis^{S_2''};$, *and* $\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}$, $\llparenthesis \mathcal{E}_1'' \rrparenthesis^{S_1''} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2'' \rrparenthesis^{S_2''}$*; or*

(o) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \texttt{return}\ \llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'};$, $\mathcal{E}_2 = \texttt{return}\ \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'};$, *and* $\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}$*; or*

(p) $S_1 \simeq_{\texttt{Low}} S_2$, $\mathcal{E}_1 = \llparenthesis o_1 \rrparenthesis^{S_1''}.\texttt{super}(\overline{\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'}})$, $\mathcal{E}_2 = \llparenthesis o_2 \rrparenthesis^{S_2''}.\texttt{super}(\overline{\llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}})$, *and* $\llparenthesis o_1 \rrparenthesis^{S_1''} \simeq_{\texttt{Low}} \llparenthesis o_2 \rrparenthesis^{S_2''}$, $\overline{\llparenthesis \mathcal{E}_1' \rrparenthesis^{S_1'}} \simeq_{\texttt{Low}} \overline{\llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}}$*; or*

(q) $S_1 \simeq_{\texttt{Low}} S_2$ *and* $\mathcal{E}_1 = \mathcal{E}_2 =;\ $ *; or*

3. *(Heaps)* $H \simeq_{\texttt{Low}} H'$ *iff the following two conditions hold:*

   (a) $\forall o$ *such that* $H(o) = \llparenthesis \texttt{new C}(\bar{\mathcal{V}}) \rrparenthesis^{S}$*, if* $S \subseteq \texttt{Low}$*, then* $H'(o) = \llparenthesis \texttt{new C}(\bar{\mathcal{V}}') \rrparenthesis^{S'}$, $S = S'$*, and* $\bar{\mathcal{V}} \simeq_{\texttt{Low}} \bar{\mathcal{V}}'$.

   (b) $\forall o$ *such that* $H'(o) = \llparenthesis \texttt{new C}(\bar{\mathcal{V}}') \rrparenthesis^{S'}$*, if* $S' \subseteq \texttt{Low}$*, then* $H(o) = \llparenthesis \texttt{new C}(\bar{\mathcal{V}}) \rrparenthesis^{S}$, $S = S'$*, and* $\bar{\mathcal{V}} \simeq_{\texttt{Low}} \bar{\mathcal{V}}'$.

4. *(Output Stream Equality).* $\omega_1(\texttt{fd},\texttt{L}_1) = \omega_2(\texttt{fd},\texttt{L}_2)$ *iff* $\omega_1(\texttt{fd},\texttt{L}_1) = \bar{\texttt{c}_1}$, $\omega_2(\texttt{fd},\texttt{L}_2) = \bar{\texttt{c}_2}$*, and* $\texttt{L}_1 = \texttt{L}_2$, $|\bar{\texttt{c}_1}| = |\bar{\texttt{c}_2}|$, $\forall i < |\bar{\texttt{c}_1}|, \texttt{c}_{1i} = \texttt{c}_{2i}$.

5. *(Input Stream Equality).* $\iota_1(\texttt{fd},\texttt{L}_1) = \iota_2(\texttt{fd},\texttt{L}_2)$ *iff* $\iota_1(\texttt{fd},\texttt{L}_1) = \bar{\texttt{c}_1}$, $\iota_2(\texttt{fd},\texttt{L}_2) = \bar{\texttt{c}_2}$*, and* $\texttt{L}_1 = \texttt{L}_2$, $|\bar{\texttt{c}_1}| = |\bar{\texttt{c}_2}|$, $\forall i < |\bar{\texttt{c}_1}|, \texttt{c}_{1i} = \texttt{c}_{2i}$.

6. *(Output Streams).* $\omega_1 \simeq_{\texttt{Low}} \omega_2$ *iff* $dom(\omega_1) = dom(\omega_2)$ *and* $\forall(\texttt{fd},\texttt{L}_1) \in dom(\omega_1)$*, either*

   (a) $\texttt{L}_1 \subseteq \texttt{Low}$, $\texttt{L}_2 \subseteq \texttt{Low}$*, and* $\omega_1(\texttt{fd},\texttt{L}_1) = \omega_2(\texttt{fd},\texttt{L}_2)$*; or*

   (b) $\texttt{L}_1 \not\subseteq \texttt{Low}$ *and* $\texttt{L}_2 \not\subseteq \texttt{Low}$.

7. *(Input Streams).* $\iota_1 \simeq_{\texttt{Low}} \iota_2$ *iff* $dom(\iota_1) = dom(\iota_2)$ *and* $\forall(\texttt{fd},\texttt{L}_1) \in dom(\iota_1)$*, either*

   (a) $\texttt{L}_1 \subseteq \texttt{Low}$, $\texttt{L}_2 \subseteq \texttt{Low}$*, and* $\iota_1(\texttt{fd},\texttt{L}_1) = \iota_2(\texttt{fd},\texttt{L}_2)$*; or*

   (b) $\texttt{L}_1 \not\subseteq \texttt{Low}$ *and* $\texttt{L}_2 \not\subseteq \texttt{Low}$.

8. $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}, H_2, \iota_2, \omega_2$ *iff* $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}$, $H_1 \simeq_{\texttt{Low}} H_2$, $\iota_1 \simeq_{\texttt{Low}} \iota_2$*, and* $\omega_1 \simeq_{\texttt{Low}} \omega_2$.

Lemma 4.16 (Properties of Bisimulation)

1. *(Reflexive)* $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1$.

2. *(Symmetric) If* $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}, H_2, \iota_2, \omega_2$*, then* $\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}, H_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1$.

3. *(Transitive) If* $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}, H_2, \iota_2, \omega_2$*, and* $\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}, H_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_3 \rrparenthesis^{S_3}, H_3, \iota_3, \omega_3$*, then* $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}_3 \rrparenthesis^{S_3}, H_3, \iota_3, \omega_3$

Proof. By induction on the structure of $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}$ and directly by Definition 4.15. □

Lemma 4.17 *If* $S \subseteq \texttt{Low}$, $S' \subseteq \texttt{Low}$, $S = S'$ *and* $H \simeq_{\texttt{Low}} H'$*, then* $newref(H, S) = newref(H', S')$

Proof. By contradiction.

Suppose $newref(H, S) = o$, $newref(H', S') = o'$, and $o \neq o'$. By the definition of $newref$, $o = loc_i^S$ and $o' = loc_{i'}^{S'}$. Now, by assumption, $S = S'$, so in order to satisfy $o \neq o'$, we must have $i \neq i'$. As per the definition of $newref$, let $i - 1$ be the largest integer, such that $loc_{i-1}^S \in dom(H)$, and let $i' - 1$ be the largest integer, such that $loc_{i'-1}^{S'} \in dom(H')$. Without loss of generality, assume $i - 1 > i' - 1$.

Now, according to Definition 4.15[3], since $S \subseteq \texttt{Low}$, and $S' \subseteq \texttt{Low}$, we must have $loc_{i-1}^S \in dom(H')$ and $loc_{i'-1}^{S'} \in dom(H)$. Since $i - 1 > i' - 1$ and $loc_{i-1}^S \in dom(H')$, then $i' - 1$ is not the largest integer such that $loc_{i'-1}^{S'} \in dom(H')$, a contradiction. Hence, the assumption that $i \neq i'$ is wrong, and therefore the assumption that $o \neq o'$ is also wrong. Hence, $o = o'$.

□

Lemma 4.18 (Value Substitution) *If* $\Gamma[\bar{\mathtt{x}} \mapsto \bar{t}], s_p, \emptyset \vdash \epsilon : \tau \backslash \mathcal{C}_e$, $\Gamma, S_p, H \vdash \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}} : \bar{\tau_v} \backslash \bar{\mathcal{C}_v}$, *and there exists* $\bar{t_l'}$ *such that for all* $\bar{t_l}$ *free in* $[s_p \mapsto S_p, \bar{t} \mapsto \bar{\tau}]\mathcal{C}_e \cup \bar{\mathcal{C}_v}$, $[\bar{t_l} \mapsto \bar{t_l'}][s_p \mapsto S_p, \bar{t} \mapsto \bar{\tau}]\mathcal{C}_e \cup \bar{\mathcal{C}_v}$ *is consistent, and* $\Gamma[\bar{\mathtt{x}} \mapsto \bar{t}], \emptyset, s_p, \mathcal{C} \vdash_{con} \epsilon : S$ *then* $\Gamma, H, S_p, \mathcal{C} \vdash_{con} [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}]\epsilon : S \cup \bar{S_v} \cup S_p$.

Proof.

Assume $\Gamma, S_p, H \vdash \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}} : \bar{\tau_v} \backslash \bar{\mathcal{C}_v}$ and $\Gamma[\bar{\mathtt{x}} \mapsto \bar{t}], s_p, \emptyset \vdash \epsilon : \tau \backslash \mathcal{C}_e$.

By Substitution Lemma 4.7, $\Gamma, S_p, H \vdash [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}]\epsilon : [\bar{t_l} \mapsto \bar{t_l'}][\bar{t} \mapsto \bar{\tau_v}, s_p \mapsto S_p]\tau \backslash [\bar{t_l} \mapsto \bar{t_l'}][\bar{t} \mapsto \bar{\tau_v}, s_p \mapsto S_p]\mathcal{C}_e$.

Let $\tau = \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle$, $[\bar{t_l} \mapsto \bar{t_l'}][\bar{t} \mapsto \bar{\tau_v}, s_p \mapsto S_p]\tau = \langle \mathcal{S}_1, \mathcal{F}_1, \mathcal{A}_1 \rangle$, $\bar{\tau_v} = \overline{\langle \mathcal{S}_v, \mathcal{F}_v, \mathcal{A}_v \rangle}$, and $\bar{t} = \overline{\langle s, f, \alpha \rangle}$.

Suppose $\Gamma, H, S_p, \mathcal{C} \vdash_{con} [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}]\epsilon : S_1$. We now show that $S_1 \subseteq S \cup S_v \cup S_p$ by showing that any $\mathtt{l} \in S_1$ is also in $S \cup S_v \cup S_p$. Let $\mathtt{l}$ be any concrete label in S_1. We have three cases according to Definition 4.1.

1. $\mathtt{l} \in \mathcal{S}_1$. We have two cases.

   (a) $\mathtt{l} \in \mathcal{S}$. Then by Definition 4.1 $\mathtt{l} \in S$, so $\mathtt{l} \in S \cup S_v \cup S_p$.

   (b) $\mathtt{l} \notin \mathcal{S}$. Then $\mathtt{l} \in \mathcal{S}_v$, so by Definition 4.1, $\mathtt{l} \in S_v$. So, $\mathtt{l} \in S \cup S_v \cup S_p$.

2. There exists an $s' \in \mathcal{S}_1$, such that $\mathtt{l} <: s' \in \mathit{Closure}(LT, \mathcal{C})$. Now, $s' \neq \bar{s}, s_p$, otherwise it would have been substituted with S_v or S_p. Hence, $s' \in \mathcal{S}$, so by Definition 4.1, $\mathtt{l} \in S$. So, $\mathtt{l} \in S \cup S_v \cup S_p$.

3. There exists an $f'.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathcal{S}_1$, such that $\mathtt{l} <: f'.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathit{Closure}(LT, \mathcal{C})$. Now, $f' \neq \bar{f}$, otherwise it would have been substituted with $\mathcal{F}_v$. Hence, $f'.\mathtt{f}.\overline{\mathsf{F}.\mathtt{f}'}.\mathsf{S} \in \mathcal{S}$, so by Definition 4.1, $\mathtt{l} \in S$. So, $\mathtt{l} \in S \cup S_v \cup S_p$.

Hence, $S_1 \subseteq S \cup S_v \cup S_p$. Then by (Sub) and Definition 4.1 $\Gamma, H, S_p, \mathcal{C} \vdash_{con} [\mathtt{x} \mapsto \llparenthesis v \rrparenthesis^{S_v}]\epsilon : S \cup S_v$. $\square$

Lemma 4.19 (Bisimulation of Substituted Values) *If* $S_v \simeq_{\mathtt{Low}} S_v'$, $\llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}} \simeq_{\mathtt{Low}} \llparenthesis \bar{v'} \rrparenthesis^{\bar{S_v'}}$, $o = o'$, $H_1 \simeq_{\mathtt{Low}} H_1'$, $\Gamma[\bar{\mathtt{x}} \mapsto \bar{t}], s_p, \emptyset \vdash \epsilon : \tau \backslash \mathcal{C}_e$, $\Gamma, S_v, H \vdash \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}} : \bar{\tau_v} \backslash \bar{\mathcal{C}_v}$, $\Gamma, S_v', H' \vdash \llparenthesis \bar{v'} \rrparenthesis^{\bar{S_v'}} : \bar{\tau_v'} \backslash \bar{\mathcal{C}_v'}$, $\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2} = \llbracket \Gamma, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\epsilon \rrbracket_e$ *there exists* $\bar{t_l}$ *such that for all* $\bar{t_i}$ *free in* $[s_p \mapsto S_v, \bar{t} \mapsto \bar{\tau_v}]\mathcal{C}_e \cup \bar{\mathcal{C}_v}$, $[\bar{t_i} \mapsto \bar{t_l}][s_p \mapsto S_v, \bar{t} \mapsto \bar{\tau_v}]\mathcal{C}_e \cup \bar{\mathcal{C}_v} \subseteq \mathcal{C}$ *and* $\mathcal{C}$ *is consistent, and there exists* $\bar{t_l'}$ *such that for all* $\bar{t_i}$ *free in* $[s_p \mapsto S_v', \bar{t} \mapsto \bar{\tau_v'}]\mathcal{C}_e \cup \bar{\mathcal{C}_v'}$, $[\bar{t_i} \mapsto \bar{t_l'}][s_p \mapsto S_v', \bar{t} \mapsto \bar{\tau_v'}]\mathcal{C}_e \cup \bar{\mathcal{C}_v'} \subseteq \mathcal{C}'$ *and* $\mathcal{C}'$ *is consistent, then there exists a translation* $\llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'} = \llbracket \Gamma', H', S_v', \mathcal{C}', [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S_v'}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S_v'}]\epsilon \rrbracket_e$, *such that* $\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2} \simeq_{\mathtt{Low}} \llparenthesis \mathcal{E}_2' \rrparenthesis^{S_2'}$.

Proof.

By induction on the structure of ϵ. We present only a few cases. The remainder follow in a similar fashion.

Case $\epsilon = \mathtt{x}$. Hence $[\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\epsilon = \llparenthesis v_i \rrparenthesis^{S_{v_i}}$ and $[\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S_v'}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S_v'}]\epsilon = \llparenthesis v_i' \rrparenthesis^{S_{v_i}'}$ and $\llparenthesis v_i \rrparenthesis^{S_{v_i}} \simeq_{\mathtt{Low}} \llparenthesis v_i' \rrparenthesis^{S_{v_i}'}$. We have two cases for v_i.

Case v_i is a constant. Then by (Val) and Definition 4.1, we have $\Gamma, H, S_v, \mathcal{C} \vdash \llparenthesis v_i \rrparenthesis^{S_{v_i}} : S_{v_i} \cup S_v$ and $\Gamma, H', S_v', \mathcal{C} \vdash \llparenthesis v_i' \rrparenthesis^{S_{v_i'}} : S_{v_i'} \cup S_v'$. Hence, by Definition 4.4, $\llparenthesis v_i \rrparenthesis^{S_{v_i} \cup S_v} = \llbracket \Gamma, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\epsilon \rrbracket_e$ and $\llparenthesis v_i' \rrparenthesis^{S_{v_i'} \cup S_v'} = \llbracket \Gamma, H', S_v', \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S_v'}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S_v'}]\epsilon \rrbracket_e$. Since $\llparenthesis v_i \rrparenthesis^{S_{v_i}} \simeq_{\mathtt{Low}} \llparenthesis v_i' \rrparenthesis^{S_{v_i}'}$ and $S_v \simeq_{\mathtt{Low}} S_v'$, we have $\llparenthesis v_i \rrparenthesis^{S_{v_i} \cup S_v} \simeq_{\mathtt{Low}} \llparenthesis v_i' \rrparenthesis^{S_{v_i}' \cup S_v'}$. The lemma follows.

Case v_i is an object identifier. Then by (Heap) and Definition 4.1, we have $\Gamma, H, S_v, \mathcal{C} \vdash \llparenthesis v_i \rrparenthesis^{S_{v_i}} : S_{v_i} \cup S_v \cup S_o$ and $\Gamma, H', S_v', \mathcal{C} \vdash \llparenthesis v_i' \rrparenthesis^{S_{v_i'}} : S_{v_i'} \cup S_v' \cup S_o$, where $H(v_i) = \llparenthesis \mathtt{new}\ \mathtt{C}(\bar{\mathcal{V}}) \rrparenthesis^{S_o}$ and $H(v_i') = \llparenthesis \mathtt{new}\ \mathtt{C}(\bar{\mathcal{V}'}) \rrparenthesis^{S_o'}$. Hence, by Definition 4.4, $\llparenthesis v_i \rrparenthesis^{S_{v_i} \cup S_v} = \llbracket \Gamma, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\epsilon \rrbracket_e$ and $\llparenthesis v_i' \rrparenthesis^{S_{v_i'} \cup S_v'} = \llbracket \Gamma, H', S_v', \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S_v'}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S_v'}]\epsilon \rrbracket_e$. We have two cases according to Definition 4.15[2a].

Subcase $S_{v_i} \subseteq \mathtt{Low}$, $S_{v_i}' \subseteq \mathtt{Low}$, $v_i = v_i'$, and $S_{v_i} = S_{v_i'}$.
Since $H_1 \simeq_{\mathtt{Low}} H_1'$, by Definition 4.15[3], $S_o \simeq_{\mathtt{Low}} S_o'$.
Since $\llparenthesis v_i \rrparenthesis^{S_{v_i}} \simeq_{\mathtt{Low}} \llparenthesis v_i' \rrparenthesis^{S_{v_i}'}$ and $S_v \simeq_{\mathtt{Low}} S_v'$, we have $\llparenthesis v_i \rrparenthesis^{S_{v_i} \cup S_v \cup S_o} \simeq_{\mathtt{Low}} \llparenthesis v_i' \rrparenthesis^{S_{v_i}' \cup S_v' \cup S_o'}$. The lemma follows.

Subcase $S_{v_i} \not\subseteq \mathtt{Low}$ and $S_{v_i}' \not\subseteq \mathtt{Low}$. Then by Definition 4.15[2a], we have $\llparenthesis v_i \rrparenthesis^{S_{v_i} \cup S_v \cup S_o} \simeq_{\mathtt{Low}} \llparenthesis v_i' \rrparenthesis^{S_{v_i}' \cup S_v' \cup S_o'}$. The lemma follows.

Case $\epsilon = \mathtt{e_a.f}$.

Hence $\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2} = \llbracket \emptyset, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\mathtt{e_a.f} \rrbracket_e$, and $\llparenthesis \mathcal{E}'_2 \rrparenthesis^{S'_2} = \llbracket \emptyset, H', S'_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S'_v}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S'_v}]\mathtt{e_a.f} \rrbracket_e$.

Suppose $[\bar{\mathtt{x}} \mapsto \bar{t}], \emptyset, s_p, \mathcal{C} \vdash_{con} [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\mathtt{e_a.f} : S_e$, then by Lemma 4.18, $\emptyset, H, S_v, \mathcal{C} \vdash_{con} [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\mathtt{e_a.f} : S \cup \bar{S_v} \cup S_v$ and $\emptyset, H', S'_v, \mathcal{C}, \vdash_{con} [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S'_v}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S'_v}]\mathtt{e_a.f} : S \cup \bar{S'_v} \cup S'_v$.

Since$\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2} = \llbracket \emptyset, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\mathtt{e_a.f} \rrbracket_e$, and $\llparenthesis \mathcal{E}'_2 \rrparenthesis^{S'_2} = \llbracket \emptyset, H', S'_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S'_v}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S'_v}]\mathtt{e_a.f} \rrbracket_e$, by Definition 4.3, $S_2 = S \cup \bar{S_v} \cup S_v$ and $S'_2 = S \cup \bar{S'_v} \cup S'_v$. Since $\bar{S_v} \simeq_{\mathtt{Low}} \bar{S'_v}$ and $S_v \simeq_{\mathtt{Low}} S'_v$, we have $S_2 \simeq_{\mathtt{Low}} S'_2$.

By Definition 4.3, $\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2} = \llparenthesis \llparenthesis \mathcal{E}_a \rrparenthesis^{S_a}.\mathtt{f} \rrparenthesis^{S_2}$, where $\llparenthesis \mathcal{E}_a \rrparenthesis^{S_a} = \llbracket \emptyset, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v} \rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis o \rrparenthesis^{S_v}]\mathtt{e_a} \rrbracket_e$, and $\llparenthesis \mathcal{E}'_2 \rrparenthesis^{S'_2} = \llparenthesis \llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a}.\mathtt{f} \rrparenthesis^{S'_2}$, where $\llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a} = \llbracket \emptyset, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis \bar{v'} \rrparenthesis^{\bar{S'_v}}, \mathtt{this} \mapsto \llparenthesis o' \rrparenthesis^{S'_v}]\mathtt{e_a} \rrbracket_e$.

By induction, $\llparenthesis \mathcal{E}_a \rrparenthesis^{S_a} \simeq_{\mathtt{Low}} \llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a}$. Since $S_2 \simeq_{\mathtt{Low}} S'_2$, by Definition 4.15[2d], $\llparenthesis \llparenthesis \mathcal{E}_a \rrparenthesis^{S_a}.\mathtt{f} \rrparenthesis^{S_2} \simeq_{\mathtt{Low}} \llparenthesis \llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a}.\mathtt{f} \rrparenthesis^{S'_2}$, that is $\llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2} \simeq_{\mathtt{Low}} \llparenthesis \mathcal{E}'_2 \rrparenthesis^{S'_2}$.

□

Lemma 4.20 (Deeply High Computation) *If $\llparenthesis \mathcal{E} \rrparenthesis^{S}, H, \iota, \omega$ terminates, $S_p \not\subseteq \mathtt{Low}$, and the label on each subconfiguration of $\llparenthesis \mathcal{E} \rrparenthesis^{S}$ contains S_p, then there exists v, H', ι', ω', such that $\llparenthesis \mathcal{E} \rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow_h^* \llparenthesis v \rrparenthesis^{S_v}, H', \iota', \omega'$ and $S_v \not\subseteq \mathtt{Low}$.*

Proof. By induction on $\rightsquigarrow_h^*$ and the structure of $\llparenthesis \mathcal{E} \rrparenthesis^{S}$ using Definition 4.13. Since the label on each subconfiguration of $\llparenthesis \mathcal{E} \rrparenthesis^{S}$ contains S_p, observing each semantic rule $\rightsquigarrow$ shows S_p must also be on the configuration of the resulting computation, and on each object reference, which by Definition 4.13 makes each step *high*. Furthermore, since S_p is on the configuration resulting from any computation, we have $S_p \subseteq S_v$, and since by assumption $S_p \not\subseteq \mathtt{Low}$, we have $S_v \not\subseteq \mathtt{Low}$. □

Lemma 4.21 (Bisimulation of High Computation)

1. *(one-step) If $\llparenthesis \mathcal{E} \rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow_h \llparenthesis \mathcal{E}' \rrparenthesis^{S'}, H', \iota', \omega'$, then $H \simeq_{\mathtt{Low}} H'$, $\iota \simeq_{\mathtt{Low}} \iota'$, and $\omega \simeq_{\mathtt{Low}} \omega'$.*

2. *(n-steps) If $\llparenthesis \mathcal{E} \rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow_h^* \llparenthesis \mathcal{E}' \rrparenthesis^{S'}, H', \iota', \omega'$, then $H \simeq_{\mathtt{Low}} H'$, $\iota \simeq_{\mathtt{Low}} \iota'$, and $\omega \simeq_{\mathtt{Low}} \omega'$.*

Proof.

1. By induction on the derivation of $\llparenthesis \mathcal{E} \rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow_h \llparenthesis \mathcal{E}' \rrparenthesis^{S'}, H', \iota', \omega'$, using Definition 4.15, Definition 4.13, and Fixed Point Lemma 4.9.

2. By induction on the length of $\rightsquigarrow_h^*$, using Lemma 4.21[1].

□

Lemma 4.22 shows that for two configurations that differ only in some high values, a low step results in the same configuration, apart from differing high values. The low portions of the heap must be the same.

Lemma 4.22 (Reduction of Low Security Configurations) *Let $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1 = \llbracket \Gamma, H_i, pc, \mathcal{C}, \mathtt{e} \rrbracket$ and $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow_l \llparenthesis \mathcal{E}_2 \rrparenthesis^{S_2}, H_2, \iota_2, \omega_2$, and $\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}, H'_1 = \llbracket \Gamma', H'_i, pc', \mathcal{C}', \mathtt{e}' \rrbracket$ and $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \simeq_{\mathtt{Low}} \llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1$, then there exists derivations $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow_l \llparenthesis \mathcal{E}''_2 \rrparenthesis^{S''_2}, H''_2, \iota_2, \omega_2$, and $\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_l \llparenthesis \mathcal{E}'''_2 \rrparenthesis^{S'''_2}, H'''_2, \iota'_2, \omega'_2$, such that $\llparenthesis \mathcal{E}''_2 \rrparenthesis^{S''_2}, H''_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \llparenthesis \mathcal{E}'''_2 \rrparenthesis^{S'''_2}, H'''_2, \iota'_2, \omega'_2$.*

Proof.
By induction on the context derivation tree of $\rightsquigarrow_l$, with case analysis on the last (bottom) reduction rule used.

Case (Field-R) Let $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1} = \llparenthesis \llparenthesis o \rrparenthesis^{S}.\mathtt{f}_i \rrparenthesis^{S_1}$ and $\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1} = \llparenthesis \llparenthesis o' \rrparenthesis^{S'}.\mathtt{f}_j \rrparenthesis^{S'_1}$. By (Field-R), $\llparenthesis \llparenthesis o \rrparenthesis^{S}.\mathtt{f}_i \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis v \rrparenthesis^{S_i \cup S \cup S_1}, H_1, \iota_1, \omega_1$, where $H(o) = \llparenthesis \mathtt{new\ C}(\bar{\mathcal{V}}) \rrparenthesis^{S_v}$ and $\mathcal{V}_i = \llparenthesis v \rrparenthesis^{S_i}$. According to Definition 4.15[2d], we have $\llparenthesis \llparenthesis o \rrparenthesis^{S}.\mathtt{f}_i \rrparenthesis^{S_1} \simeq_{\mathtt{Low}} \llparenthesis \llparenthesis o' \rrparenthesis^{S'}.\mathtt{f}_j \rrparenthesis^{S'_1}$, hence $i = j$, $S_1 \simeq_{\mathtt{Low}} S'_1$, and $\llparenthesis o \rrparenthesis^{S} \simeq_{\mathtt{Low}} \llparenthesis o' \rrparenthesis^{S'}$. Since this is a low step, by Definition 4.12 $S_i \cup S \cup S_1 \subseteq \mathtt{Low}$, hence $S_1 \subseteq \mathtt{Low}$, so by Definition 4.15[1] $S_1 = S'_1$. We have two cases according to Definition 4.15[2a].

Subcase $S \subseteq \texttt{Low}$, $S' \subseteq \texttt{Low}$, $S = S'$, and $o = o'$.

By (Field-R), $\llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'}.\texttt{f}_i \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, v' \,\rrparenthesis^{S'_i \cup S' \cup S'_1}, H'_1, \iota'_1, \omega'_1$, where $H(o) = \llparenthesis\, \texttt{new C}(\bar{\mathcal{V}}) \,\rrparenthesis^{S_v}$ and $\mathcal{V}_i = \llparenthesis\, v' \,\rrparenthesis^{S'_i}$.
By Definition 4.12 and Definition 4.15[3], $\llparenthesis\, v' \,\rrparenthesis^{S'_i} = \llparenthesis\, v \,\rrparenthesis^{S_i}$. As previously stated $S = S'$ and $S_1 = S'_1$, hence $\llparenthesis\, v \,\rrparenthesis^{S_i \cup S \cup S_1} = \llparenthesis\, v' \,\rrparenthesis^{S'_i \cup S' \cup S'_1}$, that is $\llparenthesis\, v \,\rrparenthesis^{S_i \cup S \cup S_1} \simeq_{\texttt{Low}} \llparenthesis\, v' \,\rrparenthesis^{S'_i \cup S' \cup S'_1}$. By hypothesis and Definition 4.15, $\llparenthesis\, v \,\rrparenthesis^{S_i \cup S \cup S_1}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis\, v' \,\rrparenthesis^{S'_i \cup S' \cup S'_1}, H'_1, \iota'_1, \omega'_1$

Subcase $S \not\subseteq \texttt{Low}$ and $S' \not\subseteq \texttt{Low}$.

By (Field-R), $\llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S}.\texttt{f}_i \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, v \,\rrparenthesis^{S_i \cup S \cup S_1}, H_1, \iota_1, \omega_1$. Hence $S_i \cup S \cup S_1 \not\subseteq \texttt{Low}$, so by Definition 4.12, this is not a low step, which contradicts the hypothesis, so this case cannot occur.

Case (New-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \texttt{new C}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}})^{\bar{S}} \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \texttt{new C}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}})^{\bar{S'}} \,\rrparenthesis^{S'_1}$.

By (New-R), $\llparenthesis\, \texttt{new C}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}})^{\bar{S}} \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2}; \llparenthesis\, \texttt{return}\ \llparenthesis\, o \,\rrparenthesis^{S_1} \,\rrparenthesis^{S_1}; \,\rrparenthesis^{S_1}, H_1[o \mapsto \llparenthesis\, \texttt{new C}(\llparenthesis\, \overline{\texttt{null}} \,\rrparenthesis^{\bar{S}}) \,\rrparenthesis^{S_1}], \iota_1, \omega_1$, $\mathit{cnbody}(\texttt{C}) = \texttt{super}(\bar{\texttt{e}}); \bar{\texttt{s}}$, and $\texttt{class C extends D}\ \{\ldots\}$, and

$\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2} = \llbracket\, \Gamma, H_1, S_1, \mathcal{C}, [\bar{\texttt{x}} \mapsto \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}, \texttt{this} \mapsto \llparenthesis\, o \,\rrparenthesis^{S_1}]\texttt{this.super}(\texttt{D}, \bar{\texttt{e}}); \bar{\texttt{s}} \,\rrbracket_e$.

According to Definition 4.15[2g], $\bar{S} \simeq_{\texttt{Low}} \bar{S'}$, $S_1 \simeq_{\texttt{Low}} S'_1$, and $\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}} \simeq_{\texttt{Low}} \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}$. By assumption, this is a low step, hence by Definition 4.12, $S_1 \subseteq \texttt{Low}$, and since $S_1 \simeq_{\texttt{Low}} S'_1$, by Definition 4.15[2a], $S'_1 \subseteq \texttt{Low}$, and $S_1 = S'_1$.

Assume that $\texttt{D} \neq \texttt{Object}$. The case where $\texttt{D} = \texttt{Object}$ is omitted, as it is similar, except the call to $\texttt{super}$ has no arguments.

Since we have a well-typed label table, by Constructor Typing, $\Gamma[\bar{\texttt{x}} \mapsto \bar{t}, \texttt{this} \mapsto t_t], s_p, H \vdash \bar{\texttt{s}} : \tau_c \backslash \mathcal{C}_c$ and $\Gamma[\bar{\texttt{x}} \mapsto \bar{t}, \texttt{this} \mapsto t_t], s_p, H \vdash \bar{\texttt{e}} : \bar{\tau_s} \backslash \bar{\mathcal{C}_s}$.

By assumption and (New), we have $\Gamma, pc, H_i \vdash \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}} : \bar{\tau} \backslash \bar{\mathcal{C}}$, $\Gamma, pc, H_i \vdash \llparenthesis\, \bar{o} \,\rrparenthesis^{\bar{S_1}} : \tau_o \backslash \mathcal{C}_o$, and $\Gamma', pc', H'_i \vdash \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}} : \bar{\tau'} \backslash \bar{\mathcal{C}'}$, $\Gamma', pc', H'_i \vdash \llparenthesis\, \bar{o'} \,\rrparenthesis^{\bar{S'_1}} : \tau'_o \backslash \mathcal{C}'_o$, where $\bar{\mathcal{C}}, \mathcal{C}_o, \subseteq \mathcal{C}$ and $\bar{\mathcal{C}'}, \mathcal{C}'_o \subseteq \mathcal{C}'$.

By (New) and (Method), there exists consistent constraint sets $[\bar{t_i} \mapsto \bar{t_l}][s_p \mapsto \tau_o, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau_o, t_r \mapsto t'_r](\mathcal{C}_c \cup \bar{\mathcal{C}_s} \cup \{\texttt{D.K}(\bar{\tau_s}, t_t \xrightarrow{s_p} t_r\} \cup \{\tau_c <: t_r\})$ and $[\bar{t_i} \mapsto \bar{t'_l}][s_p \mapsto \tau'_o, \bar{t} \mapsto \bar{\tau'}, t_t \mapsto \tau_o, t_r \mapsto t''_r](\mathcal{C}_c \cup \bar{\mathcal{C}_s} \cup \{\texttt{D.K}(\bar{\tau_s}, t_t \xrightarrow{s_p} t_r\} \cup \{\tau'_c <: t_r\})$.

By (New-R), $o = \mathit{newref}(H_1, S_1)$, and let $o' = \mathit{newref}(H'_1, S'_1)$. Since $S_1 \subseteq \texttt{Low}$, $S'_1 \subseteq \texttt{Low}$, $S_1 = S'_1$, and $H_1 \simeq_{\texttt{Low}} H'_1$, by Lemma 4.17 $\mathit{newref}(H_1, S_1) = \mathit{newref}(H'_1, S'_1)$, hence $o = o'$.

Since $S_1 = S'_1$, $\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}} \simeq_{\texttt{Low}} \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}$, $o = o'$, and $H_1 \simeq_{\texttt{Low}} H'_1$, by Lemma 4.19, there exists a translation $\llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2} = \llbracket\, \Gamma', H'_1, S'_1, \mathcal{C}', [\bar{\texttt{x}} \mapsto \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}, \texttt{this} \mapsto \llparenthesis\, o' \,\rrparenthesis^{S'_1}]\texttt{this.super}(\texttt{D}, \bar{\texttt{e}}); \bar{\texttt{s}} \,\rrbracket_e$. such that $\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2} \simeq_{\texttt{Low}} \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}$.

Hence, by (New-R), $\llparenthesis\, \texttt{new C}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}})^{\bar{S'}} \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}; \llparenthesis\, \texttt{return}\ \llparenthesis\, o' \,\rrparenthesis^{S'_1} \,\rrparenthesis^{S'_1}; \,\rrparenthesis^{S'_1}, H'_1[o' \mapsto \llparenthesis\, \texttt{new C}(\llparenthesis\, \overline{\texttt{null}} \,\rrparenthesis^{\bar{S'}}) \,\rrparenthesis^{S'_1}], \iota'_1, \omega'_1$.

Since $S_1 = S'_1$, $o = o'$, $S_1 \subseteq \texttt{Low}$, and $S'_1 \subseteq \texttt{Low}$, by Definition 4.15[2a], $\llparenthesis\, o \,\rrparenthesis^{S_1} \simeq_{\texttt{Low}} \llparenthesis\, o' \,\rrparenthesis^{S'_1}$, and by Definition 4.15[1] $S_1 \simeq_{\texttt{Low}} S'_1$. Hence, by Definition 4.15[2o], $\llparenthesis\, \texttt{return}\ \llparenthesis\, o \,\rrparenthesis^{S_1} \,\rrparenthesis^{S_1} \simeq_{\texttt{Low}} \llparenthesis\, \texttt{return}\ \llparenthesis\, o' \,\rrparenthesis^{S'_1} \,\rrparenthesis^{S'_1}$. Since $\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2} \simeq_{\texttt{Low}} \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}$, by Definition 4.15[2l] $\llparenthesis\, \llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2}; \llparenthesis\, \texttt{return}\ \llparenthesis\, o \,\rrparenthesis^{S_1} \,\rrparenthesis^{S_1}; \,\rrparenthesis^{S_1} \simeq_l \mathit{set} \llparenthesis\, \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}; \llparenthesis\, \texttt{return}\ \llparenthesis\, o' \,\rrparenthesis^{S'_1} \,\rrparenthesis^{S'_1}; \,\rrparenthesis^{S'_1}$.

Now, $\bar{S} \simeq_{\texttt{Low}} \bar{S'}$ implies $\llparenthesis\, \overline{\texttt{null}} \,\rrparenthesis^{\bar{S}} \simeq_{\texttt{Low}} \llparenthesis\, \overline{\texttt{null}} \,\rrparenthesis^{\bar{S'}}$ by Definition 4.15[2a]. Since $H_1 \simeq_{\texttt{Low}} H'_1$, $o = o'$, and $S_1 = S'_1$, by Definition 4.15[3] we have $H_1[o \mapsto \llparenthesis\, \texttt{new C}(\llparenthesis\, \overline{\texttt{null}} \,\rrparenthesis^{\bar{S}}) \,\rrparenthesis^{S_1}] \simeq_{\texttt{Low}} H'_1[o' \mapsto \llparenthesis\, \texttt{new C}(\llparenthesis\, \overline{\texttt{null}} \,\rrparenthesis^{\bar{S'}}) \,\rrparenthesis^{S'_1}]$. the lemma follows by Definition 4.15[8].

Case (Invoke-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_v}.\texttt{m}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}) \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_v}.\texttt{m}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}) \,\rrparenthesis^{S'_1}$. By (Invoke-R), let $\llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_v}.\texttt{m}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}) \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2 \cup S_1}, H_1, \iota_1, \omega_1$.

By Definition 4.15[2h,2a], we have two cases.

Subcase $S_v \not\subseteq \texttt{Low}$.

By Definition 4.12, this is not a low step, so this case cannot occur.

Subcase $S_v \subseteq \texttt{Low}$.

So, by Definition 4.15[2h,2a], $S_v = S'_v$ and $o = o'$.
Now, by (Invoke-R), $\mathit{mbody}(\texttt{m}, \texttt{C}) = \bar{\texttt{s}}$ and $\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2} = \llbracket\, \Gamma, H_1, S_v, \mathcal{C}, [\bar{\texttt{x}} \mapsto \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}, \texttt{this} \mapsto \llparenthesis\, o \,\rrparenthesis^{S_v}]\bar{\texttt{s}} \,\rrbracket_e$
Since we have a well-typed label table, by Method Typing, $\Gamma[\bar{\texttt{x}} \mapsto \bar{t}, \texttt{this} \mapsto t_t], s_p, H \vdash \bar{\texttt{s}} : \tau_m \backslash \mathcal{C}_m$
By assumption and (Invoke), we have $\Gamma, pc, H_i \vdash \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}} : \bar{\tau} \backslash \bar{\mathcal{C}}$, $\Gamma, pc, H_i \vdash \llparenthesis\, \bar{o} \,\rrparenthesis^{\bar{S_v}} : \tau_o \backslash \mathcal{C}_o$, and $\Gamma', pc', H'_i \vdash \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}} : \bar{\tau'} \backslash \bar{\mathcal{C}'}$, $\Gamma', pc', H'_i \vdash \llparenthesis\, \bar{o'} \,\rrparenthesis^{\bar{S'_v}} : \tau'_o \backslash \mathcal{C}'_o$, where $\bar{\mathcal{C}}, \mathcal{C}_o, \bar{\mathcal{C}'}, \mathcal{C}'_o \subseteq \mathcal{C}$.

By (Invoke) and $(Method)$, there exists consistent constraint sets $[\bar{t_i} \mapsto \bar{t_l}][s_p \mapsto \tau_o, \bar{t} \mapsto \bar{\tau}, t_t \mapsto \tau_o, t_r \mapsto t'_r](\mathcal{C}_m \cup \{\tau_m <: t_r\})$ and $[\bar{t_i} \mapsto \bar{t'_l}][s_p \mapsto \tau'_o, \bar{t} \mapsto \bar{\tau'}, t_t \mapsto \tau'_o, t_r \mapsto t''_r](\mathcal{C}_m \cup \{\tau_m <: t_r\})$.

Since $S_v = S'_v$, $\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}} \simeq_{\texttt{Low}} \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}$, $o = o'$, and $H_1 \simeq_{\texttt{Low}} H'_1$, by Lemma 4.19, there exists a translation $\llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2} = [\![\, \Gamma', H'_1, S'_v, \mathcal{C}', [\bar{\texttt{x}} \mapsto \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}, \texttt{this} \mapsto \llparenthesis\, o' \,\rrparenthesis^{S'_v}]\bar{\texttt{s}} \,]\!]_e$ such that $\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2} \simeq_{\texttt{Low}} \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}$.

Hence, by (Invoke-R), $\llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_v}.\texttt{m}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}) \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2 \cup S'_1}, H'_1, \iota'_1, \omega'_1$.

Since $S_1 \simeq_{\texttt{Low}} S'_1$, $\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2 \cup S_1} \simeq_{\texttt{Low}} \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2 \cup S'_1}$.

Since the heaps and streams are unchanged by the reduction step, the lemma follows by Definition 4.15[8].

Case (Super-R) follows a similar reasoning to (New-R) and (Invoke-R).

Case (Seq-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \llparenthesis\, v \,\rrparenthesis^{S_v}; \llparenthesis\, \bar{\mathcal{E}} \,\rrparenthesis^{\bar{S}} \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \llparenthesis\, v' \,\rrparenthesis^{S'_v}; \llparenthesis\, \bar{\mathcal{E}'} \,\rrparenthesis^{\bar{S}'} \,\rrparenthesis^{S'_1}$.

By (Seq-R), let $\llparenthesis\, \llparenthesis\, v \,\rrparenthesis^{S_v}; \llparenthesis\, \bar{\mathcal{E}} \,\rrparenthesis^{\bar{S}} \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \llparenthesis\, \bar{\mathcal{E}} \,\rrparenthesis^{\bar{S}} \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1$ and

$\llparenthesis\, \llparenthesis\, v' \,\rrparenthesis^{S'_v}; \llparenthesis\, \bar{\mathcal{E}'} \,\rrparenthesis^{\bar{S}'} \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \llparenthesis\, \bar{\mathcal{E}'} \,\rrparenthesis^{\bar{S}'} \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1$

By Definition 4.15[2l], $\llparenthesis\, \bar{\mathcal{E}} \,\rrparenthesis^{\bar{S}} \simeq_{\texttt{Low}} \llparenthesis\, \bar{\mathcal{E}'} \,\rrparenthesis^{\bar{S}'}$ and $S_1 \simeq_{\texttt{Low}} S'_1$. Hence, by Definition 4.15[2l], $\llparenthesis\, \llparenthesis\, \bar{\mathcal{E}} \,\rrparenthesis^{\bar{S}} \,\rrparenthesis^{S_1} \simeq_{\texttt{Low}} \llparenthesis\, \llparenthesis\, \bar{\mathcal{E}'} \,\rrparenthesis^{\bar{S}'} \,\rrparenthesis^{S'_1}$. Conclude with Definition 4.15[8].

Case (Assign-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_o}.\texttt{f} := \llparenthesis\, v \,\rrparenthesis^{S_v}; \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_o}.\texttt{f} := \llparenthesis\, v' \,\rrparenthesis^{S'_v}; \,\rrparenthesis^{S'_1}$. By (Assign-R), let $\llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_o}.\texttt{f} := \llparenthesis\, v \,\rrparenthesis^{S_v}; \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \texttt{null} \,\rrparenthesis^{S_1 \cup S_v \cup S_o}, H_1[o \mapsto \llparenthesis\, \texttt{new C}(\ldots, \llparenthesis\, v \,\rrparenthesis^{S_v \cup S_1 \cup S_o \cup S_i}, \ldots) \,\rrparenthesis^{S_h}], \iota_1, \omega_1$ and $\llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_o}.\texttt{f} := \llparenthesis\, v' \,\rrparenthesis^{S'_v}; \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \texttt{null} \,\rrparenthesis^{S'_1 \cup S'_v \cup S'_o}, H'_1[o' \mapsto \llparenthesis\, \texttt{new C}(\ldots, \llparenthesis\, v' \,\rrparenthesis^{S'_v \cup S'_1 \cup S'_o \cup S'_i}, \ldots) \,\rrparenthesis^{S'_h}], \iota'_1, \omega'_1$

According to Definition 4.15[2n,2a], we have two cases.

Subcase $S_o \not\subseteq \texttt{Low}$. Then by Definition 4.12, this is not a low step, so this case cannot occur.

Subcase $S_o \subseteq \texttt{Low}$. Then $o = o'$ and $S_o = S'_o$.

Again, according to Definition 4.15[2n,2a], we have two cases.

Subcase $S_v \not\subseteq \texttt{Low}$. Then by Definition 4.12, this is not a low step, so this case cannot occur.

Subcase $S_v \subseteq \texttt{Low}$. Then $v = v'$ and $S_v = S'_v$. By Definition 4.15 and assumption, we also know $S_1 \simeq_{\texttt{Low}} S'_1$, and $S_i \simeq_{\texttt{Low}} S'_i$. Hence $\llparenthesis\, v \,\rrparenthesis^{S_v \cup S_1 \cup S_o \cup S_i} \simeq_{\texttt{Low}} \llparenthesis\, v' \,\rrparenthesis^{S'_v \cup S'_1 \cup S'_o \cup S'_i}$. Therefore by Definition 4.15[3] $H_1[o \mapsto \llparenthesis\, \texttt{new C}(\ldots, \llparenthesis\, v \,\rrparenthesis^{S_v \cup S_1 \cup S_o \cup S_i}, \ldots) \,\rrparenthesis^{S_h}] \simeq_{\texttt{Low}} H'_1[o' \mapsto \llparenthesis\, \texttt{new C}(\ldots, \llparenthesis\, v' \,\rrparenthesis^{S'_v \cup S'_1 \cup S'_o \cup S'_i}, \ldots) \,\rrparenthesis^{S'_h}]$. Furthermore, $\llparenthesis\, \texttt{null} \,\rrparenthesis^{S_1 \cup S_v \cup S_o} \simeq_{\texttt{Low}} \llparenthesis\, \texttt{null} \,\rrparenthesis^{S'_1 \cup S'_v \cup S'_o}$, and the lemma follows by Definition 4.15[8].

Case (IfTrue-R)

Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \texttt{if} \ \llparenthesis\, \texttt{b} \,\rrparenthesis^{S_v} \texttt{then} \ \llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_t} \ \texttt{else} \ \llparenthesis\, \mathcal{E}_f \,\rrparenthesis^{S_f} \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \texttt{if} \ \llparenthesis\, \texttt{b}' \,\rrparenthesis^{S'_v} \texttt{then} \ \llparenthesis\, \mathcal{E}'_t \,\rrparenthesis^{S'_t} \ \texttt{else} \ \llparenthesis\, \mathcal{E}'_f \,\rrparenthesis^{S'_f} \,\rrparenthesis^{S'_1}$ and without loss of generality, assume $\texttt{b} = \texttt{True}$. So, by (IfTrue-R), $\llparenthesis\, \texttt{if} \ \llparenthesis\, \texttt{b} \,\rrparenthesis^{S_v} \texttt{then} \ \llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_t} \ \texttt{else} \ \llparenthesis\, \mathcal{E}_f \,\rrparenthesis^{S_f} \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_t}, H_1, \iota_1, \omega_1$.

By Definition 4.15[2k,2a], we have two cases

Subcase $S_v \not\subseteq \texttt{Low}$. Then by Definition 4.12, this is not a low step, so this case cannot occur.

Subcase $S_v \not\subseteq \texttt{Low}$. Then $S_v = S'_v$ and $\texttt{b} = \texttt{b}'$. Hence $\texttt{b}' = \texttt{True}$, so by (IfTrue-R),

$\llparenthesis\, \texttt{if} \ \llparenthesis\, \texttt{b}' \,\rrparenthesis^{S'_v} \texttt{then} \ \llparenthesis\, \mathcal{E}'_t \,\rrparenthesis^{S'_t} \ \texttt{else} \ \llparenthesis\, \mathcal{E}'_f \,\rrparenthesis^{S'_f} \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \mathcal{E}'_t \,\rrparenthesis^{S'_t}, H'_1, \iota'_1, \omega'_1$. By Definition 4.15[2k], $\llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_t} \simeq_{\texttt{Low}} \llparenthesis\, \mathcal{E}'_t \,\rrparenthesis^{S'_t}$. Since the heaps and streams are unchanged, the lemma follows by assumption and Definition 4.15[8].

Case (Input-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \texttt{read}_{\texttt{L}}(\llparenthesis\, \texttt{fd} \,\rrparenthesis^{S_f}) \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \texttt{read}_{\texttt{L}}(\llparenthesis\, \texttt{fd}' \,\rrparenthesis^{S'_f}) \,\rrparenthesis^{S'_1}$.

By (Input-R), let $\llparenthesis\, \texttt{read}_{\texttt{L}}(\llparenthesis\, \texttt{fd} \,\rrparenthesis^{S_f}) \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \texttt{c} \,\rrparenthesis^{\texttt{L}}, H_1, \iota_2, \omega_1$ and $\llparenthesis\, \texttt{read}_{\texttt{L}}(\llparenthesis\, \texttt{fd}' \,\rrparenthesis^{S'_f}) \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \texttt{c}' \,\rrparenthesis^{\texttt{L}}, H'_1, \iota'_2, \omega'_1$

By Definition 4.15[2i,2a], we have two cases.

Subcase $S_f \not\subseteq \texttt{Low}$. Then by premise to (Input-R), $\texttt{L} \not\subseteq \texttt{Low}$. However, Definition 4.12 requires $\texttt{L} \subseteq \texttt{Low}$ for this to be a low step. So this case cannot occur.

Subcase $S_f \subseteq \texttt{Low}$. Then $S_f = S'_f$ and $\texttt{fd} = \texttt{fd}'$. By Definition 4.12, $\texttt{L} \subseteq \texttt{Low}$, so by Definition 4.15[5], $\iota_1(\texttt{fd}, \texttt{L}) = \iota'_1(\texttt{fd}', \texttt{L})$. Hence $\texttt{c}.\iota_2(\texttt{fd}, \texttt{L}) = \texttt{c}'.\iota'_2(\texttt{fd}', \texttt{L})$, so $\texttt{c} = \texttt{c}'$ and $\iota_2(\texttt{fd}, \texttt{L}) = \iota'_2(\texttt{fd}', \texttt{L})$. Thus, by Definition 4.15[2a], $\llparenthesis\, \texttt{c} \,\rrparenthesis^{\texttt{L}} \simeq_{\texttt{Low}} \llparenthesis\, \texttt{c}' \,\rrparenthesis^{\texttt{L}}$ and by Definition 4.15[5], $\iota_2 \simeq_{\texttt{Low}} \iota'_2$. The lemma follows by Definition 4.15[8].

Case (Output-R) Let $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1} = \llparenthesis\,\texttt{write}_{\texttt{L}}(\llparenthesis\,\texttt{c}\,\rrparenthesis^{S_c}, \llparenthesis\,\texttt{fd}\,\rrparenthesis^{S_f})\,\rrparenthesis^{S_1}$ and $\llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1} = \llparenthesis\,\texttt{write}_{\texttt{L}}(\llparenthesis\,\texttt{c}'\,\rrparenthesis^{S'_c}, \llparenthesis\,\texttt{fd}'\,\rrparenthesis^{S'_f})\,\rrparenthesis^{S'_1}$.

By (Output-R), let $\llparenthesis\,\texttt{write}_{\texttt{L}}(\llparenthesis\,\texttt{c}\,\rrparenthesis^{S_c}, \llparenthesis\,\texttt{fd}\,\rrparenthesis^{S_f})\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\,\texttt{null}\,\rrparenthesis^{S_c \cup S_f \cup S_1}, H_1, \iota_1, \omega_2$ and $\llparenthesis\,\texttt{write}_{\texttt{L}}(\llparenthesis\,\texttt{c}'\,\rrparenthesis^{S'_c}, \llparenthesis\,\texttt{fd}'\,\rrparenthesis^{S'_f})\,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\,\texttt{null}\,\rrparenthesis^{S'_c \cup S'_f \cup S'_1}, H'_1, \iota'_1, \omega'_2$.

By Definition 4.15[2j,2a], we have two cases.

Subcase $S_f \not\subseteq \texttt{Low}$. By Definition 4.12 this is not a low step. So this case cannot occur.

Subcase $S_f \subseteq \texttt{Low}$. Then $S_f = S'_f$ and $\texttt{fd} = \texttt{fd}'$. Again by Definition 4.15[2j,2a], we have two cases.

Subcase $S_c \not\subseteq \texttt{Low}$. By Definition 4.12 this is not a low step. So this case cannot occur.

Subcase $S_c \subseteq \texttt{Low}$. Then $S_c = S'_c$ and $\texttt{c} = \texttt{c}'$.

We have two more cases.

Subcase $\texttt{L} \subseteq \texttt{Low}$. Then, by Definition 4.15[6], $\omega_1(\texttt{fd}, \texttt{L}) = \omega'_1(\texttt{fd}', \texttt{L})$. Since $\texttt{c} = \texttt{c}'$, we have $\texttt{c}.\omega_1(\texttt{fd}, \texttt{L}) = \texttt{c}'.\omega'_1(\texttt{fd}', \texttt{L})$. Hence by Definition 4.15[6], $\omega_2 \simeq_{\texttt{Low}} \omega'_2$. Further, since $S_c = S'_c$, $S_f = S'_f$, and $S_1 = S'_1$, by Definition 4.15[2a], $\llparenthesis\,\texttt{null}\,\rrparenthesis^{S_c \cup S_f \cup S_1} \simeq_{\texttt{Low}} \llparenthesis\,\texttt{null}\,\rrparenthesis^{S'_c \cup S'_f \cup S'_1}$. The lemma follows by Definition 4.15[8].

Subcase $\texttt{L} \not\subseteq \texttt{Low}$. Then by Definition 4.15[6], $\omega_2 \simeq_{\texttt{Low}} \omega'_2$. Since $S_c = S'_c$, $S_f = S'_f$, and $S_1 = S'_1$, by Definition 4.15[2a], $\llparenthesis\,\texttt{null}\,\rrparenthesis^{S_c \cup S_f \cup S_1} \simeq_{\texttt{Low}} \llparenthesis\,\texttt{null}\,\rrparenthesis^{S'_c \cup S'_f \cup S'_1}$. The lemma follows by Definition 4.15[8].

Case (Field-RC) Let $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1} = \llparenthesis\,\llparenthesis\,\mathcal{E}_a\,\rrparenthesis^{S_a}.\texttt{f}\,\rrparenthesis^{S_1}$ and $\llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1} = \llparenthesis\,\llparenthesis\,\mathcal{E}'_a\,\rrparenthesis^{S'_a}.\texttt{f}\,\rrparenthesis^{S'_1}$. By assumption, Definition 4.12, and (Field-RC), $\llparenthesis\,\llparenthesis\,\mathcal{E}_a\,\rrparenthesis^{S_a}.\texttt{f}\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow_l \llparenthesis\,\llparenthesis\,\mathcal{E}_b\,\rrparenthesis^{S_b}.\texttt{f}\,\rrparenthesis^{S_1}, H_2, \iota_2, \omega_2$ and $\llparenthesis\,\mathcal{E}_a\,\rrparenthesis^{S_a}, H_1, \iota_1, \omega_1 \rightsquigarrow_l \llparenthesis\,\mathcal{E}_b\,\rrparenthesis^{S_b}, H_2, \iota_2, \omega_2$. Also by (Field-RC), $\llparenthesis\,\llparenthesis\,\mathcal{E}'_a\,\rrparenthesis^{S'_a}.\texttt{f}\,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_l \llparenthesis\,\llparenthesis\,\mathcal{E}'_b\,\rrparenthesis^{S'_b}.\texttt{f}\,\rrparenthesis^{S'_1}, H'_2, \iota'_2, \omega'_2$ and $\llparenthesis\,\mathcal{E}'_a\,\rrparenthesis^{S'_a}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_l \llparenthesis\,\mathcal{E}'_b\,\rrparenthesis^{S'_b}, H'_2, \iota'_2, \omega'_2$.

By assumption and Definition 4.15[2d], $S_1 \simeq_{\texttt{Low}} S'_1$ and $\llparenthesis\,\mathcal{E}_a\,\rrparenthesis^{S_a} \simeq_{\texttt{Low}} \llparenthesis\,\mathcal{E}'_a\,\rrparenthesis^{S'_a}$. So, by Definition 4.15[8], $\llparenthesis\,\mathcal{E}_a\,\rrparenthesis^{S_a}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis\,\mathcal{E}'_a\,\rrparenthesis^{S'_a}, H'_1, \iota'_1, \omega'_1$. Hence, by induction, there exists derivations $\llparenthesis\,\mathcal{E}_a\,\rrparenthesis^{S_a}, H_1, \iota_1, \omega_1 \rightsquigarrow_l \llparenthesis\,\mathcal{E}_c\,\rrparenthesis^{S_c}, H''_2, \iota_2, \omega_2$ and $\llparenthesis\,\mathcal{E}'_a\,\rrparenthesis^{S'_a}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow_l \llparenthesis\,\mathcal{E}'_c\,\rrparenthesis^{S'_c}, H'''_2, \iota'_2, \omega'_2$, such that $\llparenthesis\,\mathcal{E}_c\,\rrparenthesis^{S_c}, H''_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \llparenthesis\,\mathcal{E}'_c\,\rrparenthesis^{S'_c}, H'''_2, \iota'_2, \omega'_2$. Then by Definition 4.15[2d], $\llparenthesis\,\llparenthesis\,\mathcal{E}_c\,\rrparenthesis^{S_c}.\texttt{f}\,\rrparenthesis^{S_1} \simeq_{\texttt{Low}} \llparenthesis\,\llparenthesis\,\mathcal{E}'_c\,\rrparenthesis^{S'_c}.\texttt{f}\,\rrparenthesis^{S'_1}$. Conclude with Definition 4.15[8], $\llparenthesis\,\llparenthesis\,\mathcal{E}_c\,\rrparenthesis^{S_c}.\texttt{f}\,\rrparenthesis^{S_1}, H''_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \llparenthesis\,\llparenthesis\,\mathcal{E}'_c\,\rrparenthesis^{S'_c}.\texttt{f}\,\rrparenthesis^{S'_1}, H'''_2, \iota'_2, \omega'_2$.

Remaining cases: (Op-R) (Cast-R) (Super-R) (Super-R') (Return-R) (Block-R) (Skip-R) (SubVal-R) (*-RC) □

Lemma 4.23 shows that for two configurations that differ only in high values, a series of high steps – where the high steps complete by either reaching a value, or being followed by a low step – result in the same configuration with possibly some of the high values substituted for other high values, and the low portions of the heap are the same.

Lemma 4.23 (Reduction of High Security Configurations) *Let $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1}, H_1 = \llbracket\,\Gamma, H_i, pc, \mathcal{C}, \texttt{e}\,\rrbracket$, and $\llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1}, H'_1 = \llbracket\,\Gamma', H'_i, pc', \mathcal{C}', \texttt{e}'\,\rrbracket$ and $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1$ and $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow^*_h \llparenthesis\,\mathcal{E}_2\,\rrparenthesis^{S_2}, H_2, \iota_2, \omega_2$, and either $\mathcal{E}_2$ is a value, or $\llparenthesis\,\mathcal{E}_2\,\rrparenthesis^{S_2}, H_2, \iota_2, \omega_2 \rightsquigarrow_l \llparenthesis\,\mathcal{E}_3\,\rrparenthesis^{S_3}, H_3, \iota_3, \omega_3$, for some $\llparenthesis\,\mathcal{E}_3\,\rrparenthesis^{S_3}, H_3, \iota_3$, and ω_3, and assume $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1$ and $\llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1$ both terminate; then there exists derivations $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow^*_h \llparenthesis\,\mathcal{E}''_2\,\rrparenthesis^{S''_2}, H''_2, \iota_2, \omega_2$, and $\llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow^*_h \llparenthesis\,\mathcal{E}'_2\,\rrparenthesis^{S'_2}, H'_2, \iota'_2, \omega'_2$, such that $\llparenthesis\,\mathcal{E}''_2\,\rrparenthesis^{S''_2}, H''_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \llparenthesis\,\mathcal{E}'_2\,\rrparenthesis^{S'_2}, H'_2, \iota'_2, \omega'_2$.*

Proof.

By induction on length of the reduction $\rightsquigarrow^*_h$, and the height of the reduction derivation tree.

Base Case (reflexive): $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1} = \llparenthesis\,\mathcal{E}_2\,\rrparenthesis^{S_2}$, $H_1 = H_2$, $\iota_1 = \iota_2$, and $\omega_1 = \omega_2$. Then, by reflexivity of $\rightsquigarrow^*$, $\llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1} = \llparenthesis\,\mathcal{E}'_2\,\rrparenthesis^{S'_2}$, $H'_1 = H'_2$, $\iota'_1 = \iota'_2$, and $\omega'_1 = \omega'_2$. The lemma follows by assumption, $\llparenthesis\,\mathcal{E}_2\,\rrparenthesis^{S_2}, H_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \llparenthesis\,\mathcal{E}'_2\,\rrparenthesis^{S'_2}, H'_2, \iota'_2, \omega'_2$.

Inductive Case: $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow_h \llparenthesis\,\mathcal{E}\,\rrparenthesis^{S}, H, \iota, \omega \rightsquigarrow^*_h \llparenthesis\,\mathcal{E}_2\,\rrparenthesis^{S_2}, H_2, \iota_2, \omega_2$.

By induction on the context derivation tree of $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow_h \llparenthesis\,\mathcal{E}\,\rrparenthesis^{S}, H, \iota, \omega$, with case analysis on the last (bottom) reduction rule used.

Case (Field-R) Let $\llparenthesis\,\mathcal{E}_1\,\rrparenthesis^{S_1} = \llparenthesis\,\llparenthesis\,o\,\rrparenthesis^{S_o}.\texttt{f}_i\,\rrparenthesis^{S_1}$ and $\llparenthesis\,\mathcal{E}'_1\,\rrparenthesis^{S'_1} = \llparenthesis\,\llparenthesis\,o'\,\rrparenthesis^{S'_o}.\texttt{f}_i\,\rrparenthesis^{S'_1}$. By (Field-R), let $\llparenthesis\,\llparenthesis\,o\,\rrparenthesis^{S_o}.\texttt{f}_i\,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\,v\,\rrparenthesis^{S_i \cup S_1 \cup S_o}, H_1, \iota_1, \omega_1$ and $\llparenthesis\,\llparenthesis\,o'\,\rrparenthesis^{S'_o}.\texttt{f}_i\,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\,v'\,\rrparenthesis^{S'_i \cup S'_1 \cup S'_o}, H'_1, \iota'_1, \omega'_1$.

By Definition 4.13, $S_i \cup S_1 \cup S_o \not\subseteq \texttt{Low}$. By Definition 4.15[2d, 2a, 3], we have $S_i \simeq_{\texttt{Low}} S'_i$, $S_1 \simeq_{\texttt{Low}} S'_1$, and $S_o \simeq_{\texttt{Low}} S'_o$. Hence $S'_i \cup S'_1 \cup S'_o \not\subseteq \texttt{Low}$. Then, by Definition 4.15[2a], $\llparenthesis\,v\,\rrparenthesis^{S_i \cup S_1 \cup S_o} \simeq_{\texttt{Low}} \llparenthesis\,v'\,\rrparenthesis^{S'_i \cup S'_1 \cup S'_o}$. The lemma follows by Definition 4.15[8].

Case (Invoke-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_v} \mathtt{.m}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}) \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_v} \mathtt{.m}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}) \,\rrparenthesis^{S'_1}$. By (Invoke-R), let $\llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_v} \mathtt{.m}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}) \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \mathcal{E}_a \,\rrparenthesis^{S_1 \cup S_a}, H_1, \iota_1, \omega_1$.

By Definition 4.13, we have $S_v \not\subseteq \mathtt{Low}$. Hence by Definition 4.15[2h, 2a] and assumption, $S'_v \not\subseteq \mathtt{Low}$.

By assumption, $\llparenthesis\, \mathcal{E}_a \,\rrparenthesis^{S_1 \cup S_a}, S_v, H_1, \iota_1, \omega_1$ terminates, and we have just shown that $S_v \not\subseteq \mathtt{Low}$. By (Invoke-R), $\llparenthesis\, \mathcal{E}_a \,\rrparenthesis^{S_a} = [\![\Gamma, H, S_v, \mathcal{C}, [\bar{\mathtt{x}} \mapsto \llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}}, \mathtt{this} \mapsto \llparenthesis\, o \,\rrparenthesis^{S_v}]\bar{\mathtt{s}}]\!]_e$, so by Lemma 4.5 and Definition 4.4, S_v is on every subconfiguration of $\llparenthesis\, \mathcal{E}_a \,\rrparenthesis^{S_a}$. Hence by Lemma 4.20, $\llparenthesis\, \mathcal{E}_a \,\rrparenthesis^{S_1 \cup S_a}, S_v, H_1, \iota_1, \omega_1 \rightsquigarrow^*_h \llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2}, H_2, \iota_2, \omega_2$, where $\mathcal{E}_2 = v_2$ for some v_2, and $S_2 \not\subseteq \mathtt{Low}$. By Lemma 4.21[2], $H_1 \simeq_{\mathtt{Low}} H_2$, $\iota_1 \simeq_{\mathtt{Low}} \iota_2$, and $\omega_1 \simeq_{\mathtt{Low}} \omega_2$.

Similarly, by (Invoke-R), let $\llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_v} \mathtt{.m}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}) \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \mathcal{E}'_a \,\rrparenthesis^{S'_1 \cup S'_a}, H'_1, \iota'_1, \omega'_1$. By assumption $\llparenthesis\, \mathcal{E}'_a \,\rrparenthesis^{S'_1 \cup S'_a}, S'_v, H'_1, \iota'_1, \omega'_1$ terminates, and we have shown that $S'_v \not\subseteq \mathtt{Low}$. By (Invoke-R), $\llparenthesis\, \mathcal{E}'_a \,\rrparenthesis^{S'_a} = [\![\Gamma', H', S'_v, \mathcal{C}', [\bar{\mathtt{x}} \mapsto \llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}}, \mathtt{this} \mapsto \llparenthesis\, o' \,\rrparenthesis^{S'_v}]\bar{\mathtt{s}}]\!]_e$, so by Lemma 4.5 and Definition 4.4, S'_v is on every subconfiguration of $\llparenthesis\, \mathcal{E}'_a \,\rrparenthesis^{S'_a}$. (Regardless of which actual typing is used, S'_v, as the program counter, will occur on every subconfiguration. (Sub) can only add labels, in essence making things more *high*, which is inconsequential.) Hence by Lemma 4.20, $\llparenthesis\, \mathcal{E}'_a \,\rrparenthesis^{S'_1 \cup S'_a}, S'_v, H'_1, \iota'_1, \omega'_1 \rightsquigarrow^*_h \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}, H'_2, \iota'_2, \omega'_2$, where $\mathcal{E}'_2 = v'_2$ for some v'_2, and $S'_2 \not\subseteq \mathtt{Low}$. By Lemma 4.21[2], $H'_1 \simeq_{\mathtt{Low}} H'_2$, $\iota'_1 \simeq_{\mathtt{Low}} \iota'_2$ and $\omega'_1 \simeq_{\mathtt{Low}} \omega'_2$.

Now, since $S_2 \not\subseteq \mathtt{Low}$, $S'_2 \not\subseteq \mathtt{Low}$, $\mathcal{E}_2 = v_2$, and $\mathcal{E}'_2 = v'_2$, by Definition 4.15[2], $\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2} \simeq_{\mathtt{Low}} \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}$. By Lemma 4.16[2,3], we have $H_2 \simeq_{\mathtt{Low}} H'_2$, $\iota_2 \simeq_{\mathtt{Low}} \iota'_2$ and $\omega_2 \simeq_{\mathtt{Low}} \omega'_2$. Then by Definition 4.15[8], $\llparenthesis\, \mathcal{E}_2 \,\rrparenthesis^{S_2}, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \llparenthesis\, \mathcal{E}'_2 \,\rrparenthesis^{S'_2}, H'_2, \iota'_2, \omega'_2$.

Case (New-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \mathtt{new\ C}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}})^{\bar{S}} \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \mathtt{new\ C}(\llparenthesis\, \bar{v'} \,\rrparenthesis^{\bar{S'_v}})^{\bar{S}'} \,\rrparenthesis^{S'_1}$. By (New-R), let
$\llparenthesis\, \mathtt{new\ C}(\llparenthesis\, \bar{v} \,\rrparenthesis^{\bar{S_v}})^{\bar{S}} \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \llparenthesis\, \mathcal{E}_a \,\rrparenthesis^{S_a}; \llparenthesis\, \mathtt{return}\ \llparenthesis\, o \,\rrparenthesis^{S_1} \,\rrparenthesis^{S_1}; \,\rrparenthesis^{S_1}, H_a, \iota_1, \omega_1$ and $H_a = H_1[o \mapsto \llparenthesis\, \mathtt{new\ C}(\llparenthesis\, \overline{\mathtt{null}} \,\rrparenthesis^{\bar{S}}) \,\rrparenthesis^{S_1}]$.

By Definition 4.13, $S_1 \not\subseteq \mathtt{Low}$, and by Definition 4.15[2g], $S'_1 \not\subseteq \mathtt{Low}$. Now, we have $newref(H_1, S_1) = o = loc_i^{S_1}$ and $newref(H'_1, S'_1) = o' = loc_j^{S'_1}$. So, if $o \in dom(H'_1)$, then by the definition of $newref$, $H'_1(o) = \llparenthesis\, \mathtt{new\ C}(\llparenthesis\, \overline{\mathcal{V}} \,\rrparenthesis^{\bar{S_\mathtt{v}}}) \,\rrparenthesis^{S_1}$. Similarly, if $o' \in dom(H_1)$, then by the definition of $newref$, $H_1(o') = \llparenthesis\, \mathtt{new\ C}(\llparenthesis\, \overline{\mathcal{V}'} \,\rrparenthesis^{\bar{S'_\mathtt{v}}}) \,\rrparenthesis^{S'_1}$. Since $S_1 \not\subseteq \mathtt{Low}$ and $S'_1 \not\subseteq \mathtt{Low}$, by Definition 4.15[3], $H_a \simeq_{\mathtt{Low}} H'_a$.

The remaining argument follows the (Invoke-R) case.

Case (Super-R) follows a similar reasoning to (Invoke-R).

Case (IfTrue-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \mathtt{if}\ \ \llparenthesis\, \mathtt{b} \,\rrparenthesis^{S_v} \mathtt{then}\ \llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_t}\ \mathtt{else}\ \llparenthesis\, \mathcal{E}_f \,\rrparenthesis^{S_f} \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \mathtt{if}\ \ \llparenthesis\, \mathtt{b}' \,\rrparenthesis^{S'_v} \mathtt{then}\ \llparenthesis\, \mathcal{E}'_t \,\rrparenthesis^{S'_t}\ \mathtt{else}\ \llparenthesis\, \mathcal{E}'_f \,\rrparenthesis^{S'_f} \,\rrparenthesis^{S'_1}$ and without loss of generality, assume $\mathtt{b} = \mathtt{True}$. So, by (IfTrue-R), $\llparenthesis\, \mathtt{if}\ \ \llparenthesis\, \mathtt{b} \,\rrparenthesis^{S_v} \mathtt{then}\ \llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_t}\ \mathtt{else}\ \llparenthesis\, \mathcal{E}_f \,\rrparenthesis^{S_f} \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_t}, H_1, \iota_1, \omega_1$.

We have two subcases:

Subcase $S_v \subseteq \mathtt{Low}$

Then, by Definition 4.15[2k,2a], $\mathtt{b}' = \mathtt{b} = \mathtt{True}$. Hence, by (IfTrue-R), $\llparenthesis\, \mathtt{if}\ \ \llparenthesis\, \mathtt{b}' \,\rrparenthesis^{S'_v} \mathtt{then}\ \llparenthesis\, \mathcal{E}'_t \,\rrparenthesis^{S'_t}\ \mathtt{else}\ \llparenthesis\, \mathcal{E}'_f \,\rrparenthesis^{S'_f} \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \mathcal{E}'_t \,\rrparenthesis^{S'_t}, H'_1, \iota'_1, \omega'_1$. The lemma follows by induction on $\llparenthesis\, \mathcal{E}_t \,\rrparenthesis^{S_v \cup S_t}, H_1, \iota_1, \omega_1$.

Subcase $S_v \not\subseteq \mathtt{Low}$

According to assumption and Lemma 4.5, each subconfiguration in $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1}$ must contain the labels in S_v. Since $S_v \not\subseteq \mathtt{Low}$, the label on each subconfiguration is also not a subset of $\mathtt{Low}$ (the label is the *high*). The lemma follows with the same reasoning as the (Invoke-R) case.

Case (Assign-R) Let $\llparenthesis\, \mathcal{E}_1 \,\rrparenthesis^{S_1} = \llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_o}\mathtt{.f} := \llparenthesis\, v \,\rrparenthesis^{S_v}; \,\rrparenthesis^{S_1}$ and $\llparenthesis\, \mathcal{E}'_1 \,\rrparenthesis^{S'_1} = \llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_o}\mathtt{.f} := \llparenthesis\, v' \,\rrparenthesis^{S'_v}; \,\rrparenthesis^{S'_1}$. Since both executions terminate, By (Assign-R) and Fixed Point Lemma 4.9, there exists reductions $\llparenthesis\, \llparenthesis\, o \,\rrparenthesis^{S_o}\mathtt{.f} := \llparenthesis\, v \,\rrparenthesis^{S_v}; \,\rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis\, \mathtt{null} \,\rrparenthesis^{S_1 \cup S_v \cup S_o}, H_2, \iota_1, \omega_1$ and $\llparenthesis\, \llparenthesis\, o' \,\rrparenthesis^{S'_o}\mathtt{.f} := \llparenthesis\, v' \,\rrparenthesis^{S'_v}; \,\rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis\, \mathtt{null} \,\rrparenthesis^{S'_1 \cup S'_v \cup S'_o}, H'_2, \iota'_1, \omega'_1$, where $H_2 = H_1[o \mapsto \llparenthesis\, \mathtt{new\ C}(\ldots, \llparenthesis\, v \,\rrparenthesis^{S_v \cup S_1 \cup S_o \cup S_i}, \ldots) \,\rrparenthesis^{S_h}]$ and $H'_2 = H'_1[o' \mapsto \llparenthesis\, \mathtt{new\ C}(\ldots, \llparenthesis\, v' \,\rrparenthesis^{S'_v \cup S'_1 \cup S'_o \cup S'_i}, \ldots) \,\rrparenthesis^{S'_h}]$, such that $S_v \cup S_1 \cup S_o \cup S_i = S_i$ and $S'_v \cup S'_1 \cup S'_o \cup S'_i = S'_i$.

By Definition 4.13, $S_1 \cup S_v \cup S_o \not\subseteq \mathtt{Low}$. Since by Definition 4.15[2d,2a], $S_1 \simeq_{\mathtt{Low}} S'_1$, $S_v \simeq_{\mathtt{Low}} S'_v$, and $S_o \simeq_{\mathtt{Low}} S'_o$, we have $S'_1 \cup S'_v \cup S'_o \not\subseteq \mathtt{Low}$. Hence, by Definition 4.15[2a], $\llparenthesis\, \mathtt{null} \,\rrparenthesis^{S_1 \cup S_v \cup S_o} \simeq_{\mathtt{Low}} \llparenthesis\, \mathtt{null} \,\rrparenthesis^{S'_1 \cup S'_v \cup S'_o}$.

It remains to be shown that $H_2 \simeq_{\mathtt{Low}} H'_2$. $S_1 \cup S_v \cup S_o \not\subseteq \mathtt{Low}$ implies $S_v \cup S_1 \cup S_o \cup S_i \not\subseteq \mathtt{Low}$ and $S'_1 \cup S'_v \cup S'_o \not\subseteq \mathtt{Low}$ implies $S'_v \cup S'_1 \cup S'_o \cup S'_i \not\subseteq \mathtt{Low}$. Since $S_v \cup S_1 \cup S_o \cup S_i = S_i$ and $S'_v \cup S'_1 \cup S'_o \cup S'_i = S'_i$, we have $S_i \not\subseteq \mathtt{Low}$ and $S'_i \not\subseteq \mathtt{Low}$.

According to Definition 4.15[2n,2a], we have two cases.

Subcase $S_o \not\subseteq \texttt{Low}$. Then $S'_o \not\subseteq \texttt{Low}$.

According to Definition 4.15[3], we must satisfy two cases to show $H_2 \simeq_{\texttt{Low}} H'_2$.

Subcase Definition 4.15[3a], if $o \in dom(H'_1)$.

Let $H_1(o) = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v_i \rrparenthesis^{S_i}, \ldots) \rrparenthesis^{S_h}$ and $H'_1(o) = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v''_i \rrparenthesis^{S''_i}, \ldots) \rrparenthesis^{S''_h}$. Thus, $H_2(o) = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v \rrparenthesis^{S_v \cup S_1 \cup S_o \cup S_i}, \ldots) \rrparenthesis^{S_h}$ and $H'_2(o) = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v''_i \rrparenthesis^{S''_i}, \ldots) \rrparenthesis^{S''_h}$.

Suppose $S'_h \not\subseteq \texttt{Low}$. Then by Definition 4.15[3], $S_h \not\subseteq \texttt{Low}$, and this case is satisfied.

Suppose instead that $S'_h \subseteq \texttt{Low}$. Then by Definition 4.15[3], $S_h = S'_h$ and $\llparenthesis v_i \rrparenthesis^{S_i} \simeq_{\texttt{Low}} \llparenthesis v''_i \rrparenthesis^{S''_i}$. Since $S_v \cup S_1 \cup S_o \cup S_i \not\subseteq \texttt{Low}$ and $S_i \not\subseteq \texttt{Low}$, by Definition 4.15[2a], $\llparenthesis v \rrparenthesis^{S_v \cup S_1 \cup S_o \cup S_i} \simeq_{\texttt{Low}} \llparenthesis v''_i \rrparenthesis^{S''_i}$, and the case is satisfied.

Subcase Definition 4.15[3b], if $o' \in dom(H_1)$.

Let $H_1(o') = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v'''_i \rrparenthesis^{S'''_i}, \ldots) \rrparenthesis^{S'''_h}$ and $H'_1(o') = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v'_i \rrparenthesis^{S'_i}, \ldots) \rrparenthesis^{S'_h}$. Thus, $H_2(o') = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v'''_i \rrparenthesis^{S'''_i}, \ldots) \rrparenthesis^{S'''_h}$ and $H'_2(o') = \llparenthesis \texttt{new C}(\ldots, \llparenthesis v' \rrparenthesis^{S'_v \cup S'_1 \cup S'_o \cup S'_i}, \ldots) \rrparenthesis^{S'_h}$.

Suppose $S'_h \not\subseteq \texttt{Low}$. Then by Definition 4.15[3], $S'''_h \not\subseteq \texttt{Low}$, and this case is satisfied.

Suppose instead that $S'_h \subseteq \texttt{Low}$. Then by Definition 4.15[3], $S'''_h = S'_h$ and $\llparenthesis v'''_i \rrparenthesis^{S'''_i} \simeq_{\texttt{Low}} \llparenthesis v'_i \rrparenthesis^{S'_i}$. Since $S'_v \cup S'_1 \cup S'_o \cup S'_i \not\subseteq \texttt{Low}$ and $S'_i \not\subseteq \texttt{Low}$, by Definition 4.15[2a], $\llparenthesis v'''_i \rrparenthesis^{S'''_i} \simeq_{\texttt{Low}} \llparenthesis v' \rrparenthesis^{S'_v \cup S'_1 \cup S'_o \cup S'_i}$, and the case is satisfied.

Therefore, by Definition 4.15[3], $H_2 \simeq_{\texttt{Low}} H'_2$.

Subcase $S_o \subseteq \texttt{Low}$. Then $o = o'$ and $S_o = S'_o$.

Hence, by Definition 4.15[2a], $\llparenthesis v \rrparenthesis^{S_v \cup S_1 \cup S_o \cup S_i} \simeq_{\texttt{Low}} \llparenthesis v' \rrparenthesis^{S'_v \cup S'_1 \cup S'_o \cup S'_i}$. Therefore, since $o = o'$, by assumption $H_1 \simeq_{\texttt{Low}} H'_1$, and Definition 4.15[3], $H_2 \simeq_{\texttt{Low}} H'_2$.

The lemma follows by Definition 4.15[8].

Case (Seq-R) Let $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1} = \llparenthesis \llparenthesis v \rrparenthesis^{S_v}; \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}} \rrparenthesis^{S_1}$ and $\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1} = \llparenthesis \llparenthesis v' \rrparenthesis^{S'_v}; \llparenthesis \bar{\mathcal{E}}' \rrparenthesis^{\bar{S}'} \rrparenthesis^{S'_1}$.

By (Seq-R), let $\llparenthesis \llparenthesis v \rrparenthesis^{S_v}; \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}} \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}} \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1$ and $\llparenthesis \llparenthesis v' \rrparenthesis^{S'_v}; \llparenthesis \bar{\mathcal{E}}' \rrparenthesis^{\bar{S}'} \rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis \llparenthesis \bar{\mathcal{E}}' \rrparenthesis^{\bar{S}'} \rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1$

By Definition 4.15[2l], $\llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}} \simeq_{\texttt{Low}} \llparenthesis \bar{\mathcal{E}}' \rrparenthesis^{\bar{S}'}$ and $S_1 \simeq_{\texttt{Low}} S'_1$. Hence, by Definition 4.15[2l], $\llparenthesis \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}} \rrparenthesis^{S_1} \simeq_{\texttt{Low}} \llparenthesis \llparenthesis \bar{\mathcal{E}}' \rrparenthesis^{\bar{S}'} \rrparenthesis^{S'_1}$. The lemma follows by induction on $\llparenthesis \llparenthesis \bar{\mathcal{E}} \rrparenthesis^{\bar{S}} \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1$.

Case (Input-R) Let $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1} = \llparenthesis \texttt{read}_{\texttt{L}}(\llparenthesis \texttt{fd} \rrparenthesis^{S_f}) \rrparenthesis^{S_1}$ and $\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1} = \llparenthesis \texttt{read}_{\texttt{L}}(\llparenthesis \texttt{fd}' \rrparenthesis^{S'_f}) \rrparenthesis^{S'_1}$.

By (Input-R), let $\llparenthesis \texttt{read}_{\texttt{L}}(\llparenthesis \texttt{fd} \rrparenthesis^{S_f}) \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis \texttt{c} \rrparenthesis^{\texttt{L}}, H_1, \iota_2, \omega_1$, and $\iota_1(\texttt{fd}, \texttt{L}) = \texttt{c}.\iota_2(\texttt{fd}, \texttt{L})$,

$\llparenthesis \texttt{read}_{\texttt{L}}(\llparenthesis \texttt{fd}' \rrparenthesis^{S'_f}) \rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis \texttt{c}' \rrparenthesis^{\texttt{L}}, H'_1, \iota'_2, \omega'_1$, and $\iota'_1(\texttt{fd}', \texttt{L}) = \texttt{c}'.\iota'_2(\texttt{fd}', \texttt{L})$.

By Definition 4.13, $\texttt{L} \not\subseteq \texttt{Low}$. Hence, by Definition 4.15[7], $\iota_2 \simeq_{\texttt{Low}} \iota'_2$ and by Definition 4.15[2a], $\llparenthesis \texttt{c} \rrparenthesis^{\texttt{L}} \simeq_{\texttt{Low}} \llparenthesis \texttt{c}' \rrparenthesis^{\texttt{L}}$. The lemma follows by Definition 4.15[8].

Case (Output-R) Let $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1} = \llparenthesis \texttt{write}_{\texttt{L}}(\llparenthesis \texttt{c} \rrparenthesis^{S_c}, \llparenthesis \texttt{fd} \rrparenthesis^{S_f}) \rrparenthesis^{S_1}$ and $\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1} = \llparenthesis \texttt{write}_{\texttt{L}}(\llparenthesis \texttt{c}' \rrparenthesis^{S'_c}, \llparenthesis \texttt{fd}' \rrparenthesis^{S'_f}) \rrparenthesis^{S'_1}$.

By (Output-R), let $\llparenthesis \texttt{write}_{\texttt{L}}(\llparenthesis \texttt{c} \rrparenthesis^{S_c}, \llparenthesis \texttt{fd} \rrparenthesis^{S_f}) \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow \llparenthesis \texttt{null} \rrparenthesis^{S_c \cup S_f \cup S_1}, H_1, \iota_1, \omega_2$, and $\omega_2(\texttt{fd}, \texttt{L}) = \texttt{c}.\omega_1(\texttt{fd}, \texttt{L})$,

$\llparenthesis \texttt{write}_{\texttt{L}}(\llparenthesis \texttt{c}' \rrparenthesis^{S'_c}, \llparenthesis \texttt{fd}' \rrparenthesis^{S'_f}) \rrparenthesis^{S'_1}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow \llparenthesis \texttt{null} \rrparenthesis^{S'_c \cup S'_f \cup S'_1}, H'_1, \iota'_1, \omega'_2$, and $\omega'_2(\texttt{fd}', \texttt{L}) = \texttt{c}'.\omega'_1(\texttt{fd}', \texttt{L})$.

By premise to (Output-R), $S_c \cup S_f \cup S_1 \subseteq \texttt{L}$. By Definition 4.13, $S_c \cup S_f \cup S_1 \not\subseteq \texttt{Low}$. Hence, $\texttt{L} \not\subseteq \texttt{Low}$, and by Definition 4.15[6], $\omega_2 \simeq_{\texttt{Low}} \omega'_2$.

By Definition 4.15[2j,2a], $S_c \simeq_{\texttt{Low}} S'_c$, $S_f \simeq_{\texttt{Low}} S'_f$, and $S_1 \simeq_{\texttt{Low}} S'_1$, so $S'_c \cup S'_f \cup S'_1 \not\subseteq \texttt{Low}$. Hence $\llparenthesis \texttt{null} \rrparenthesis^{S_c \cup S_f \cup S_1} \simeq_{\texttt{Low}} \llparenthesis \texttt{null} \rrparenthesis^{S'_c \cup S'_f \cup S'_1}$. The lemma follows by Definition 4.15[8].

Case (Field-RC) Let $\llparenthesis \mathcal{E}_1 \rrparenthesis^{S_1} = \llparenthesis \llparenthesis \mathcal{E}_a \rrparenthesis^{S_a}.\texttt{f} \rrparenthesis^{S_1}$ and $\llparenthesis \mathcal{E}'_1 \rrparenthesis^{S'_1} = \llparenthesis \llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a}.\texttt{f} \rrparenthesis^{S'_1}$.

By assumption, Definition 4.13, and (Field-RC), $\llparenthesis \llparenthesis \mathcal{E}_a \rrparenthesis^{S_a}.\texttt{f} \rrparenthesis^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow^*_h \llparenthesis \llparenthesis \mathcal{E}_b \rrparenthesis^{S_b}.\texttt{f} \rrparenthesis^{S_1}, H_2, \iota_2, \omega_2$ and

$\llparenthesis \mathcal{E}_a \rrparenthesis^{S_a}, H_1, \iota_1, \omega_1 \rightsquigarrow^*_h \llparenthesis \mathcal{E}_b \rrparenthesis^{S_b}, H_2, \iota_2, \omega_2$. By assumption and Definition 4.15[2d], $S_1 \simeq_{\texttt{Low}} S'_1$ and $\llparenthesis \mathcal{E}_a \rrparenthesis^{S_a} \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a}$. So, by Definition 4.15[8], $\llparenthesis \mathcal{E}_a \rrparenthesis^{S_a}, H_1, \iota_1, \omega_1 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a}, H'_1, \iota'_1, \omega'_1$. Hence, by induction, there exists derivations $\llparenthesis \mathcal{E}_a \rrparenthesis^{S_a}, H_1, \iota_1, \omega_1 \rightsquigarrow^*_h \llparenthesis \mathcal{E}_c \rrparenthesis^{S_c}, H''_2, \iota_2, \omega_2$ and $\llparenthesis \mathcal{E}'_a \rrparenthesis^{S'_a}, H'_1, \iota'_1, \omega'_1 \rightsquigarrow^*_h \llparenthesis \mathcal{E}'_c \rrparenthesis^{S'_c}, H'''_2, \iota'_2, \omega'_2$, such that $\llparenthesis \mathcal{E}_c \rrparenthesis^{S_c}, H''_2, \iota_2, \omega_2 \simeq_{\texttt{Low}} \llparenthesis \mathcal{E}'_c \rrparenthesis^{S'_c}, H'''_2, \iota'_2, \omega'_2$. Then by Definition 4.15[2d], $\llparenthesis \llparenthesis \mathcal{E}_c \rrparenthesis^{S_c}.\texttt{f} \rrparenthesis^{S_1} \simeq_{\texttt{Low}} \llparenthesis \llparenthesis \mathcal{E}'_c \rrparenthesis^{S'_c}.\texttt{f} \rrparenthesis^{S'_1}$. By Definition 4.15[8], $\llparenthesis \llparenthesis \mathcal{E}_c \rrparenthesis^{S_c}.\texttt{f} \rrparenthesis^{S_1}, H''_2, \iota_2, \omega_2 \simeq_{\texttt{Lo}}$ $\llparenthesis \llparenthesis \mathcal{E}'_c \rrparenthesis^{S'_c}.\texttt{f} \rrparenthesis^{S'_1}, H'''_2, \iota'_2, \omega'_2$.

Now, if $\llparenthesis \llparenthesis \mathcal{E}_c \rrparenthesis^{S_c}.\texttt{f} \rrparenthesis^{S_1}, H''_2, \iota_2, \omega_2 \rightsquigarrow_l \llparenthesis \mathcal{E}_3 \rrparenthesis^{S_3}, H_3, \iota_3, \omega_3$, then this case is complete. Otherwise, if $\llparenthesis \llparenthesis \mathcal{E}_c \rrparenthesis^{S_c}.\texttt{f} \rrparenthesis^{S_1}, H''_2, \iota_2, \omega_2 \rightsquigarrow^*_h \llparenthesis \mathcal{E}_3 \rrparenthesis^{S_3}, H_3, \iota_3, \omega_3$, conclude by induction on $\rightsquigarrow^*_h$.

Remaining cases: (Op-R) (Cast-R) (Super-R') (Return-R) (Block-R) (Skip-R) (SubVal-R) (*-RC) $\Box$

The noninterference theorem assumes the initial input streams are bisimilar with respect to `Low`. In other words, the user with access only to data labeled by a subset of `Low` cannot observe anything about data with labels that are not in `Low` (i.e. *high*). Initial output streams are also assumed to be bisimilar with respect to `Low`, although for normal programs, they will initially be empty. The theorem assumes the input streams are fixed before execution, but applies to *any* stream of inputs.

Theorem 4.24 (Non-deterministic Noninterference) *Suppose* $\emptyset, \emptyset, \emptyset \vdash \bar{\mathtt{s}} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$, *and* $\iota_1 \simeq_{\mathtt{Low}} \iota_1'$, *and* $\omega_1 \simeq_{\mathtt{Low}} \omega_1'$, *and* $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1, \omega_1$ *and* $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1', \omega_1'$ *both terminate then there exists derivations* $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1, \omega_1 \leadsto^* \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c}, H_2, \iota_2, \omega_2$ *and* $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1', \omega_1' \leadsto^* \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'}, H_2', \iota_2', \omega_2'$, *such that* $\iota_2 \simeq_{\mathtt{Low}} \iota_2'$ *and* $\omega_2 \simeq_{\mathtt{Low}} \omega_2'$. *(and* $\llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c} \simeq_{\mathtt{Low}} \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'}$*)*

Proof.

The theorem follows by repeated use of lemmas 4.23 and 4.22 (allowed by Translation lemma 4.10). The reduction $\leadsto^*$ consists of alternating $\leadsto_h^*$ and $\leadsto_l$ steps. The number of high steps in $\leadsto_h^*$ can also be zero. Since $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1, \omega_0$ terminates, we have $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1, \omega_0 \leadsto^* \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S}, H_2, \iota_2, \omega_2$, which can be written as $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1, \omega_0 \leadsto_h^* \llparenthesis\, \mathcal{E}_3'' \,\rrparenthesis^{S_3''}, H_3'', \iota_3, \omega_3 \leadsto_l \llparenthesis\, \mathcal{E}_4'' \,\rrparenthesis^{S_4''}, H_4'', \iota_4, \omega_4 \leadsto_h^* \ldots \leadsto_h^* \llparenthesis\, \mathtt{c}'' \,\rrparenthesis^{S_c''}, H_2'', \iota_2, \omega_2$.

Then, according to Lemmas 4.23 and 4.22, there exists derivations $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1, \omega_0 \leadsto_h^* \llparenthesis\, \mathcal{E}_3 \,\rrparenthesis^{S_3}, H_3, \iota_3, \omega_3 \leadsto_l \llparenthesis\, \mathcal{E}_4 \,\rrparenthesis^{S_4}, H_4, \iota_4, \omega_4 \leadsto_h^* \ldots \leadsto_h^* \llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c}, H_2, \iota_2, \omega_2$ and $\llbracket \emptyset, \emptyset, \emptyset, \mathcal{C}, \bar{\mathtt{s}} \rrbracket, \iota_1, \omega_0 \leadsto_h^* \llparenthesis\, \mathcal{E}_3' \,\rrparenthesis^{S_3'}, H_3', \iota_3, \omega_3 \leadsto_l \llparenthesis\, \mathcal{E}_4' \,\rrparenthesis^{S_4'}, H_4', \iota_4, \omega_4 \leadsto_h^* \ldots \leadsto_h^* \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'}, H_2', \iota_2, \omega_2$, such that $\llparenthesis\, \mathcal{E}_3 \,\rrparenthesis^{S_3}, H_3, \iota_3, \omega_3 \simeq_{\mathtt{Low}} \llparenthesis\, \mathcal{E}_3' \,\rrparenthesis^{S_3'}, H_3', \iota_3, \omega_3$, $\llparenthesis\, \mathcal{E}_4 \,\rrparenthesis^{S_4}, H_4, \iota_4, \omega_4 \simeq_{\mathtt{Low}} \llparenthesis\, \mathcal{E}_4' \,\rrparenthesis^{S_4'}, H_4', \iota_4, \omega_4$, and so forth, until $\llparenthesis\, \mathtt{c} \,\rrparenthesis^{S_c}, H_2, \iota_2, \omega_2 \simeq_{\mathtt{Low}} \llparenthesis\, \mathtt{c}' \,\rrparenthesis^{S_c'}, H_2', \iota_2, \omega_2$.

Hence, regardless of whether the execution ends in a low or high step, lemmas 4.23 and 4.22 both conclude that the low input and output streams remain equivalent: $\omega_2 \simeq_{\mathtt{Low}} \omega_2'$, and $\iota_2 \simeq_{\mathtt{Low}} \iota_2'$, as do the final values: $\mathtt{c} = \mathtt{c}'$. The theorem follows. $\Box$

4.5 Unlabeled Semantics and Noninterference

We now describe an unlabeled semantics with a deterministic evaluation relation, and show that it's execution is equivalent in terms of values to the labeled semantics, and therefore the noninterference result holds for this semantics. The deterministic evaluation relation $\rightarrow$, defined for unlabeled expressions and statements is identical to the semantic definitions of the labeled configuration semantic relation $\leadsto$, only without the labels. Reductions are of the form $\epsilon, M_1', \iota_1, \omega_1 \rightarrow \epsilon', M_2', \iota_2, \omega_2$, where M is defined identical to H, only lacking labels. The reduction rules are presented in Figure 11 and Figure 12. The rules for reductions under context are similar to those in the labeled semantics, and are therefore omitted. In the deterministic semantics, $\mathtt{e}$ is defined as in the original grammar in Figure 1, with the following addition: $\mathtt{e} ::= \ldots \mid o \mid \mathit{IOErr} \mid o.\mathtt{super}(\mathtt{C}, \bar{\mathtt{e}})$.

The bisimulation in Definition 4.25 shows the relationship between labeled configurations and unlabeled expressions and statements. The latter are the same as the former, only without the labels. Further, the heaps are the same only M lacks labels, modulo exact heap locations.

Definition 4.25 (Bisimulation of Labeled and Unlabeled)

1. *(Configurations and Expressions). For a partial function* $\beta : \{\bar{o}\} \rightharpoonup \{\bar{o'}\}$, *then* $\epsilon \sim_\beta \llparenthesis\, \mathcal{E} \,\rrparenthesis^{S}$ *iff either*

   (a) $\epsilon = \mathcal{E} = \mathtt{CO}$*; or*

   (b) $\epsilon = \mathcal{E} = \mathit{IOErr}$*; or*

   (c) $\epsilon = o$ *and* $\mathcal{E} = \beta(o)$*; or*

   (d) $\epsilon = \mathcal{E}_1 = \mathtt{x}$*, for some* $\mathtt{x}$*; or*

   (e) $\epsilon = \mathcal{E} = \mathtt{this}$*; or*

   (f) $\epsilon = \epsilon'.\mathtt{f}$, $\mathcal{E} = \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}.\mathtt{f}$*, and* $\epsilon' \sim_\beta \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}$*; or*

   (g) $\epsilon = (\mathtt{C})\ \epsilon'$, $\mathcal{E} = (\mathtt{C})\ \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}$*, and* $\epsilon' \sim_\beta \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}$*; or*

   (h) $\epsilon = \epsilon' \oplus \epsilon''$, $\mathcal{E} = \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'} \oplus \llparenthesis\, \mathcal{E}'' \,\rrparenthesis^{S''}$*, and* $\epsilon' \sim_\beta \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}$, $\epsilon'' \sim_\beta \llparenthesis\, \mathcal{E}'' \,\rrparenthesis^{S''}$*; or*

   (i) $\epsilon = \mathtt{new\ C}(\overline{\epsilon'})$, $\mathcal{E}_2 = \mathtt{new\ C}(\overline{\llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}})^{\bar{S}''}$*, and* $\overline{\epsilon'} \sim_\beta \overline{\llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}}$*; or*

   (j) $\epsilon = \epsilon'.\mathtt{m}(\overline{\epsilon''})$, $\mathcal{E} = \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}.\mathtt{m}(\overline{\llparenthesis\, \mathcal{E}'' \,\rrparenthesis^{S''}})$*, and* $\epsilon' \sim_\beta \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}$, $\overline{\epsilon''} \sim_\beta \overline{\llparenthesis\, \mathcal{E}'' \,\rrparenthesis^{S''}}$*; or*

(Field-R)	$o.\mathtt{f}_i, M, \iota, \omega \rightarrow v_i, M, \iota, \omega$	where $\mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\,\bar{\mathtt{f}}$, and $M(o) = \mathtt{new\ C}(\bar{v})$
(Op-R)	$\mathtt{c} \oplus \mathtt{c}', M, \iota, \omega \rightarrow v, M, \iota, \omega$	where $v = \mathtt{c} \oplus \mathtt{c}'$
(Cast-R)	$(\mathtt{D})\ o, M, \iota, \omega \rightarrow o, M, \iota, \omega$	where $\mathtt{C} <: \mathtt{D}$
(Declassify-R)	$\mathtt{Declassify}(v, \mathtt{L}), M, \iota, \omega \rightarrow v, M, \iota, \omega$	
(New-R)	$\mathtt{new\ C}(\bar{v}), M, \iota, \omega \rightarrow \bar{\mathtt{s}}'; \mathtt{return}\ o, M', \iota, \omega$ where $\mathit{cnbody}(\mathtt{C}) = \mathtt{super}(\bar{\mathtt{e}}); \bar{\mathtt{s}}$ and $\mathtt{class\ C\ extends\ D}\ \{\dots\}$ and $\bar{\mathtt{s}}' = [\bar{\mathtt{x}} \mapsto \bar{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \bar{\mathtt{e}}); \bar{\mathtt{s}}]$ and $M' = M[o \mapsto \mathtt{new\ C}(\overline{\mathtt{null}})]$and $o = \mathit{newref}(M)$	
(Invoke-R)	$o.\mathtt{m}(\bar{v}), M, \iota, \omega \rightarrow \bar{\mathtt{s}}', M, \iota, \omega$ where $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{s}}$ and $\bar{\mathtt{s}}' = [\bar{\mathtt{x}} \mapsto \bar{v}, \mathtt{this} \mapsto o]\bar{\mathtt{s}}$	
(Super-R)	$o.\mathtt{super}(\mathtt{C}, \bar{v}), M, \iota, \omega \rightarrow \bar{\mathtt{s}}', M, \iota, \omega$ where $\mathit{cnbody}(\mathtt{C}) = \mathtt{super}(\bar{\mathtt{e}}); \bar{\mathtt{s}}$ and $\mathtt{class\ C\ extends\ D}\ \{\dots\}$ and $\bar{\mathtt{s}}' = [\bar{\mathtt{x}} \mapsto \bar{v}, \mathtt{this} \mapsto o]\mathtt{this.super}(\mathtt{D}, \bar{\mathtt{e}}); \bar{\mathtt{s}}$	
(Super-R')	$o.\mathtt{super}(\mathtt{Object}), M, \iota, \omega \rightarrow \mathtt{null}, M, \iota, \omega$	
(IfTrue-R)	$\mathtt{if\ True\ then\ e_1\ else\ e_2}, M, \iota, \omega \rightarrow \mathtt{e}_1, M, \iota, \omega$	
(Seq-R)	$v; \bar{\mathtt{e}}, M, \iota, \omega \rightarrow \bar{\mathtt{e}}, M, \iota, \omega$	
(Return-R)	$\mathtt{return}\ v;, M, \iota, \omega \rightarrow v, M, \iota, \omega$	
(Block-R)	$\{\mathtt{e}\}, M, \iota, \omega \rightarrow \mathtt{e}, M, \iota, \omega$	
(Skip-R)	$;, M, \iota, \omega \rightarrow \mathtt{null}, M, \iota, \omega$	
(Assign-R)	$o.\mathtt{f} := v;, M, \iota, \omega \rightarrow \mathtt{null}, M[o \mapsto \mathtt{new\ C}(\dots, v, \dots)], \iota, \omega$	where $\mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\,\bar{\mathtt{f}}$ and $M(o) = \mathtt{new\ C}(\bar{v})$
newref(M) = $o = \mathit{loc}_i$ where $i - 1$ is the largest integer, such that $\mathit{loc}_{i-1} \in M$		

Figure 11: Operational Semantics Reduction Rules

(Input-R)	$\mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\mathtt{fd}), M, \iota, \omega \rightarrow \mathtt{c}, M, \iota', \omega$	where $\mathtt{L} = S_i$ and $\mathtt{L}' = I_i$ and $\iota(\mathtt{fd}, S_i, I_i) = \mathtt{c}.\iota'(\mathtt{fd}, S_i, I_i)$
(Output-R)	$\mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\mathtt{c}, \mathtt{fd}), M, \iota, \omega \rightarrow \mathtt{null}, M, \iota, \omega'$	where $\mathtt{L} = S_i$ and $\mathtt{L}' = I_i$ and $\omega'(\mathtt{fd}, S_i, I_i) = \mathtt{c}.\omega(\mathtt{fd}, S_i, I_i)$
(InErr-R)	$\mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\mathtt{fd}), M, \iota, \omega \rightarrow \mathit{IOErr}, M, \iota, \omega$	where $\mathtt{L} \neq S_i$ or $\mathtt{L}' \neq I_i$ and $\iota(\mathtt{fd}, S_i, I_i)$
(OutErr-R)	$\mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\mathtt{c}, \mathtt{fd}), M, \iota, \omega \rightarrow \mathit{IOErr}, M, \iota, \omega$	where $\mathtt{L} \neq S_i$ or $\mathtt{L}' \neq I_i$ and $\omega(\mathtt{fd}, S_i, I_i)$
(IOErr-R)	$\epsilon, M, \iota, \omega \rightarrow \mathit{IOErr}, M, \iota, \omega$	where IOErr is a subexpression of ϵ

Figure 12: Operational Semantics IO Reduction Rules

(k) $\epsilon = \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\epsilon')$, $\mathcal{E} = \mathtt{read}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'})$*, and* $\epsilon' \sim_\beta \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}$*; or*

(l) $\epsilon = \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\epsilon', \epsilon'')$, $\mathcal{E} = \mathtt{write}_{(\mathtt{L},\mathtt{L}')}(\llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}, \llparenthesis\, \mathcal{E}'' \,\rrparenthesis^{S''})$*, and* $\epsilon' \sim_\beta \llparenthesis\, \mathcal{E}' \,\rrparenthesis^{S'}$, $\epsilon'' \sim_\beta \llparenthesis\, \mathcal{E}'' \,\rrparenthesis^{S''}$*; or*

(m) $\epsilon = \texttt{if}\ \epsilon'\ \texttt{then}\ \epsilon''\ \texttt{else}\ (\!| \mathcal{E}_1''' |\!)^{S_1'''}$, $\mathcal{E} = \texttt{if}\ (\!| \mathcal{E}' |\!)^{S'}\ \texttt{then}\ (\!| \mathcal{E}'' |\!)^{S''}\ \texttt{else}\ (\!| \mathcal{E}''' |\!)^{S'''}$, *and* $\epsilon' \sim_\beta (\!| \mathcal{E}' |\!)^{S'}$, $\epsilon'' \sim_\beta (\!| \mathcal{E}'' |\!)^{S''}$, $(\!| \mathcal{E}_1''' |\!)^{S_1'''} \sim_\beta (\!| \mathcal{E}''' |\!)^{S'''}$*; or*

(n) $\epsilon = \overline{\epsilon'}$, $\mathcal{E} = \overline{(\!| \mathcal{E}' |\!)^{S'}}$, *and* $\overline{\epsilon'} \sim_\beta \overline{(\!| \mathcal{E}' |\!)^{S'}}$*; or*

(o) $\epsilon = \{\epsilon'\}$, $\mathcal{E} = \{(\!| \mathcal{E}' |\!)^{S'}\}$, *and* $\epsilon' \sim_\beta (\!| \mathcal{E}' |\!)^{S'}$*; or*

(p) $\epsilon = \epsilon'.\texttt{f} := \epsilon'';$, $\mathcal{E} = (\!| \mathcal{E}' |\!)^{S'}.\texttt{f} := (\!| \mathcal{E}'' |\!)^{S''};$, *and* $\epsilon' \sim_\beta (\!| \mathcal{E}' |\!)^{S'}$, $\epsilon'' \sim_\beta (\!| \mathcal{E}'' |\!)^{S''}$*; or*

(q) $\epsilon = \texttt{return}\ \epsilon';$, $\mathcal{E} = \texttt{return}\ (\!| \mathcal{E}' |\!)^{S'};$, *and* $\epsilon' \sim_\beta (\!| \mathcal{E}' |\!)^{S'}$*; or*

(r) $\epsilon = (\!| o_1 |\!)^{S_1''}.\texttt{super}(\overline{\epsilon'})$, $\mathcal{E} = (\!| o |\!)^{S''}.\texttt{super}(\overline{(\!| \mathcal{E}' |\!)^{S'}})$, *and* $(\!| o_1 |\!)^{S_1''} \sim_\beta (\!| o |\!)^{S''}$, $\overline{\epsilon'} \sim_\beta \overline{(\!| \mathcal{E}' |\!)^{S'}}$*; or*

(s) $\epsilon = \mathcal{E} = ;$ *; or*

2. *(Heaps)* $M \sim_\beta H$ *iff the following two conditions hold:*

   (a) β *is a bijection between* $dom(\beta)$ *and* $rng(\beta)$*.*

   (b) $dom(\beta) = dom(M)$ *and* $rng(\beta) = dom(H)$

   (c) $\forall o \in dom(M)$*, if* $M(o) = \texttt{new C}(\bar{v})$*, then* $H(\beta(o)) = (\!| \texttt{new C}(\bar{\mathcal{V}}) |\!)^S$ *and* $\bar{v} \sim_\beta \bar{\mathcal{V}}$*.*

We prove the equivalence of the evaluations of the labeled and unlabeled semantics in the following lemma.

Lemma 4.26 (Evaluation Equivalence) *If* $\epsilon, M_1, \iota_1, \omega_1 \rightarrow \epsilon', M_2, \iota_2, \omega_2$*, and* $(\!| \mathcal{E}_1 |\!)^{S_1}, H_1 = [\![\Gamma, H_i, pc, \mathcal{C}, \epsilon]\!]$*, and* $\epsilon \sim_\beta (\!| \mathcal{E}_1 |\!)^{S_1}$*, and* $H_1 \sim_\beta M_1$*, then there exists* β' *where* $\beta \subseteq \beta'$*, such that* $(\!| \mathcal{E}_1 |\!)^{S_1}, H_1, \iota_1, \omega_1 \rightsquigarrow (\!| \mathcal{E}_2 |\!)^{S_2}, H_2, \iota_2, \omega_2$*, and* $\epsilon' \sim_\beta (\!| \mathcal{E}_2 |\!)^{S_2}$*, and* $H_1 \sim_\beta M_1$*.*

Proof. By induction on the structure of ϵ and using Soundness Theorem 4.11. □

The use of the soundness theorem is necessary since the labeled semantics has an a rule that may reduce a configuration to CkFail. However, Soundness Theorem 4.11 shows this will not happen for a translated expression.

This further means that in the following Noninterference result for the unlabeled semantics, when we assume termination we are assuming only the usual types of non-termination do not occur, since there is no check failure in the semantics. In other words, all reads and writes that are not IOErr will occur, but the following Corollary ensures the low IO streams remain equivalent.

Corollary 4.27 (Noninterference) *Suppose* $\emptyset, \emptyset, \emptyset \vdash \bar{\texttt{s}} : \langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle \backslash \mathcal{C}$*, and* $\iota_1 \simeq_{\texttt{Low}} \iota_1'$*, and* $\omega_1 \simeq_{\texttt{Low}} \omega_1'$*, and* $\bar{\texttt{s}}, \iota_1, \omega_1 \rightarrow^* \texttt{c}, H_2, \iota_2, \omega_2$ *and* $\bar{\texttt{s}}, \iota_1', \omega_1' \rightarrow^* \texttt{c}', H_2', \iota_2', \omega_2'$*, then* $\iota_2 \simeq_{\texttt{Low}} \iota_2'$ *and* $\omega_2 \simeq_{\texttt{Low}} \omega_2'$*, and* $\texttt{c} = \texttt{c}'$*.*

Proof. Directly by Theorem 4.24 and Lemma 4.26. □

5 Top-level Policies

In this section, we present a system for declaring class-based policies at the top level of a program, meaning the policy will not be buried in the code. This also provides a simpler means of adding information flow controls to programs, since the underlying programs will not need to include any explicit flow annotations and so there is no need to define a new language syntax for an information flow extension.

We use a simple translation-based approach for these top-level policies. Given a valid program and a top-level policy, the translation produces a new program with security levels on `read` and `write` expressions of `InputStream` and `OutputStream` subclasses, and `Declassify` (and `Endorse`) statements on method return values, when downgrading (or upgrading) is warranted. Policies are declared at the per-method level in a class. Each policy statement for a class `C` produces a translation, where method `M` is translated to `M′`, which includes the information flow statement.

Policies consist of four types of statements. `read` policies declare the sets of security labels for an input channel using the Java representation of an `InputStream` subclass, `C`. Hence, the `read` method of `C` is re-written to perform a low level read operation with the security labels given by the policy. In a similar manner, `write` policies declare the sets of security labels for an output channel using the Java representation of an `OutputStream` subclass. The `write` method is re-written to perform a low level write operation with the security labels given by the policy. Notice we require both the `InputStream` and `OutputStream` subclasses to have a file descriptor as a (private) field. While the abstract classes `InputStream` and

$$\begin{array}{l}
\texttt{class C} : (S,I) \quad \text{where } \texttt{C} <: \texttt{InputStream} \text{ and } \texttt{FileDescriptor fd} \text{ is a (private) field of } \texttt{C}. \\
\quad \texttt{int read()}\ \{\bar{\texttt{s}}\} \qquad\qquad \Rightarrow \quad \texttt{int read()}\ \{\texttt{return read}_{(S,I)}(\texttt{fd});\} \\
\\
\texttt{class C} : (S,I) \quad \text{where } \texttt{C} <: \texttt{OutputStream} \text{ and } \texttt{FileDescriptor fd} \text{ is a (private) field of } \texttt{C}. \\
\quad \texttt{void write(e)}\ \{\bar{\texttt{s}}\} \qquad \Rightarrow \quad \texttt{void write(e)}\ \{\texttt{write}_{(S,I)}(\texttt{e},\texttt{fd});\} \\
\\
\texttt{class C},\ \texttt{method RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}}) : \texttt{Declassify(L)} \quad \text{where } \texttt{RT} \neq \texttt{void} \\
\quad \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return e};\} \quad \Rightarrow \quad \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return Declassify(e,L)};\} \\
\\
\texttt{class C},\ \texttt{method RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}}) : \texttt{Endorse(L)} \quad \text{where } \texttt{RT} \neq \texttt{void} \\
\quad \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return e};\} \quad \Rightarrow \quad \texttt{RT m}(\bar{\texttt{C}}\ \bar{\texttt{x}})\{\bar{\texttt{s}};\ \texttt{return Endorse(e,L)};\}
\end{array}$$

Figure 13: Top-level Policy Translation

`OutputStream` do not have such a requirement, usable stream classes do, such as `FileInputStream`. In the Java implementation, low-level reads and writes are actually native methods. It is these low-level methods that we are re-defining. (In actuality, there is some variation in the Java implementations of various `Stream` classes. For example, `FileInputStream` uses an additional private native method `readBytes` for low-level reads of multiple bytes. For complete Java, we would need to define additional translations to satisfy these inconsistencies, though the policy format will remain the same.)

Any sub-classes of `InputStream` and `OutputStream` that do not have a defined policy receive the default policy, described earlier. Hence all unspecified input streams are low secrecy and low integrity; the default policy for an output stream is also low secrecy and low integrity.

`Declassify` statements specify what labels will be declassified from a method's return value. Note that although we provide the ability to specify declassification policies at the top-level, declassification of data requires knowledge of the underlying code to be sure the data is truly diluted enough to warrant declassification, so it must be used with care. `Declassify` statements can only be applied to methods with non-`void` return types, since it is the value that is returned from the method that is declassified. `Endorse` statements are defined analogously for integrity upgrading. Note that MJ requires that methods only have one return statement, at the end of the method body. Generalizing the language to other return statements requires the translation to be applied to any return statement within a method body.

5.1 Example Top-level Policies

The following is a top-level policy for the program for changing passwords in section 2.2.

```
class SysFileIS:
 ({high,sys},{high,sys})
class UserIS
 read():  Endorse({high})
class PwdFileOS:
 ({high,sys},{high})
class PwdFile
 ChangePwd(String uname,oldpwd,newpwd):
  Declassify({high,sys})
```

Supposing the program of Section 2.2 had all of the explicit information flow labels, checks, and declassifications removed, to give a regular Java program; if the above policy were then applied to that stripped program, we would obtain the program presented in Section 2.2 all over again. Even though programs may contain no explicit information flow policy information, it still may be necessary to rewrite parts of a program for purposes of adding a fine-grained information flow policy: a unique subclass needs to be defined for each different IO security policy. This in fact can be viewed as a good step, because it leads to a more object-oriented information flow policy. The problem is that most programs use a limited number of classes for input and output operations. For example, file reads will use `FileInputStream`, regardless of the sensitivity of the file being read.

5.2 Policies at Code Deployment Time

One of the drawbacks of other information flow systems is the inflexibility of their security policy definitions. Many type annotations are required, and security levels are added directly to the code. The result is that the programmer is the one defining the security policy, not the users deploying the code in their specific setting – different deployment settings will have different security requirements for the same program.

By contrast, our top-level policies are definable at deployment time, since whomever is defining the policy need not have intimate knowledge of the source code. A drawback to our method is that a new static analysis must be done for each set of policies. Although performance is less of an issue for a static analysis, we would still like to avoid long run-times, especially if the policy needs tweaking. We can avert this in the following manner. Variables for labels are automatically generated to represent input and output channels when typing classes and the program body. The constraint closure is then computed with these variables. A user-defined policy describes the security level each channel represents, and each variable is replaced by this security level in the closed constraint set. All that remains is a simple consistency check of SC and IC constraints, which will be much faster, since the majority of the cost of the analysis is due to closing the constraint set.

6 Related Work

Static analysis of information flow control systems is a well-studied area [13, 31, 32, 4, 1, 23]; Sabelfeld and Myers present a survey in [26]. Much of the literature focuses on proving formal results for small programming languages, though there has been some effort to define working systems. Flow Caml [24] is an information flow extension to Core ML. The Jif system provides information flow control for full Java [18, 19].

O'Neill *et. al.* described an information flow security model for interactive IO using a simple imperative language [21]. They demonstrate that a simple type system can be used to obtain noninterference in an interactive setting involving user strategies, then expand the model to incorporate nondeterministic choice. Instead of dealing with user strategies directly, our noninterference result accounts for *any* possible set of input streams a user may define, which includes all strategies that a user may employ. We do not, however, provide nondeterminism, which is a significant portion of their paper. In comparison, our system provides security for Middleweight Java, a much larger language. This allows their type system to be much simpler, although they do not describe an IO-based inference mechanism, as we do, and polymorphism is not a concern since their language does not allow methods. To our knowledge, this is the only other information flow type system that formally models interactive IO. All other systems consider only a batch input and output model, where all inputs are available prior to program execution, and outputs are only available upon completion.

Jif [18, 19] is unique as an information flow system since it covers essentially the full Java language, but it lacks a formal analysis. Checks on IO channels are intermixed with the multitude of other internal checks within a program (e.g. on function application, or assignment). Our system is designed to reduce the number of checks to IO points only. Jif provides parametric polymorphism and some inference of labels. Programs must be annotated with security labels, including label parameters for polymorphic classes. This creates a backward compatibility issue, where all code must be re-coded to introduce the proper annotations. Additionally, method overloading requires subclass types to conform to the types of the superclass.

In contrast, our type system infers all label types and parametric types, removing the need for additional program annotations. Our label types are inferred for existing code, meaning libraries can be used as is, provided the proper labels and checks are placed on the IO points in the program. Our concrete class analysis [3, 22, 33] tracks the concrete classes of objects through the program, allowing us to statically determine a conservative approximation of the runtime object. This means overridden methods in the subclass can have different types from the superclass, and the type system will correctly distinguish the information flow controls on the different objects statically.

Banerjee and Naumann [4, 5] prove a batch-model noninterference property for an information flow type system for a Java-like language using a denotational semantics. They provide an inference extension for libraries that are parameterized by security levels [29]. This form of polymorphism resembles Jif's, requiring annotations in the form of label parameters. They also require polymorphic types for methods must be satisfied by all overriding methods. As mentioned above, we employ a more implicit polymorphism that requires no program modifications, and we prove soundness and interactive noninterference using an extensible operational approach.

Flow Caml [24] provides label type inference and parametric polymorphism for an information flow extension to Core ML. They prove soundness of type inference and a batch-model noninterference property. Our type system is significantly different, since it is based on an object-oriented language, which presents unique issues, (i.e. inheritance and dynamic dispatch) that do not arise in a functional language.

Several works have developed policies for downgrading data. One approach is for the labels to contain downgrading policies which describe when it is safe to declassify the data, whether after a certain method call, operation, or some other property [16, 9]. In comparison, our policies for downgrading (and upgrading) are attached to the methods, similar to Hicks *et. al.*'s notion of declassifiers [14]. The method policies describe what labels will be downgraded for data passed to the method. This mechanism follows the object-oriented philosophy, allowing downgrading at the class and method level, and showing it in the API.

7 Conclusion

We have presented a static information flow type inference system for Middleweight Java and formally proved its correctness. Our type system provides a high level of polymorphism to promote IO-based policies and code re-use in multiple security contexts. We provide a top-level policy description, which automatically inserts information flow controls in a program and clarifies the policy in the API. Changes to Java programs are therefore minor, as only the underlying IO operations change. Type inference and easily identifiable policies are a necessity for a usable information flow system.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] French Security Incident Response Team Advisories. Veritas backup exec and netbackup remote file access vulnerability. `http://www.frsirt.com/english/advisories/2005/1387`, August 2005.

[3] Ole Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *LNCS*. SV, 1995.

[4] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, 2002.

[5] A. Banerjee and D. Naumann. Using access control for secure information flow in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169. IEEE Computer Society Press, 2003., 2003.

[6] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, The MITRE Corporation, Bedford, MA, 1975.

[7] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, Massachusetts, April 1977.

[8] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[9] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[10] Symantec Corp. Symantec brightmail antispam static database password. `http://securityresponse.symantec.com/avcenter/security/Content/2005.05.31a.html`, June 2005.

[11] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[12] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[13] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[14] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification:: high-level policy for a security-typed language. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74, New York, NY, USA, 2006. ACM Press.

[15] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *Workshop on Formal Aspects in Security and Trust (FAST)*, Sep. 2003.

[16] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[17] Peng Li and Steve Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Foundations of Computer Security Workshop (FCS)*, 2005.

[18] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[19] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999.

[20] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[21] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[22] John Plevyak and Andrew Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 324–340, 1994.

[23] Franois Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 46–57, Montral, Canada, 2000.

[24] Franois Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.

[25] Paul Roberts. Cisco warns of wireless security hole. *Computerworld*, April 2004. `http://www.computerworld.com/securitytopics/security/holes/story/0,10801,92015,00.html`.

[26] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Jounal on Selected Areas in Communications*, 21(1), January 2003.

[27] Scott Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 382–396. Springer Verlag, March 2000.

[28] Scott F. Smith and Mark Thober. Towards usable information flow security in java. Technical report, Johns Hopkins University, March 2007. `http://www.cs.jhu.edu/~mthober/`.

[29] Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proc. of the Eleventh International Static Analysis Symposium (SAS)*, volume 3148, pages 84–99. Lecture Notes in Computer Science, Springer-Verlag, August 2004.

[30] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

[31] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[32] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

[33] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *European Conference on Object-Oriented Programming(ECOOP'01)*, Budapest, Hungary, June 2001.

[34] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, page 5, Washington, DC, USA, 2001. IEEE Computer Society.