Secure Information Flow by Access Control: A Security Type System of Dual-Access Labels

Omitted for submission[1]

1 Omitted for submission

Abstract

Programming languages such as Java and C# execute code with different levels of trust in the same process, and rely on a fine-grained access control model for users to manage the security requirements of program code from different sources. While such a security model is simple enough to be used in practice to protect systems from many hostile programs downloaded over a network, it does not guard against information-based attacks, such as confidentiality and integrity violations.

We introduce a novel security model, called *Dual-Access Label* (*DAL*), to capture information-based security requirements of programs written in these languages. DAL labels extend the access control model by specifying both the accessibility and capability of program code, and use them to constrain information flows between code from different sources. Accessibility specifies the privileges necessary to access the code while capability indicates the privileges held by the code. DAL's security policy places a two-way obligation on both ends of information flow so that they must have sufficient capability to meet the accessibility of each other.

Unlike traditional lattice-based security models, our security model offers more flexible information flow relations induced by the security policy that does not have to be transitive. It provides both confidentiality and integrity guarantees while allowing cyclic information flows among code with different security labels, as desired in many applications. We present a generic security type system to enforce possibly intransitive information flow polices, including DAL, statically at compile-time. Such security type system provides a new notion of intransitive noninterference that generalizes the standard notion of transitive noninterference in lattice-based security models.

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Modern programming languages, such as Java and .NET Common Language Runtime (CLR), have been designed for Internet applications and extensible systems. In order to execute code with different levels of trust in the same process, these languages rely on a fine-grained access control model to manage the security requirements of program code from different sources. Typically, security-sensitive code (e.g. code accessing system resources) is encapsulated by the library code. While untrusted code (e.g. code downloaded from the Internet) may use the library code, only authorized code is allowed to access security-sensitive code either directly or indirectly. Such a security model provides an intransitive access control policy to prevent unwanted transitive accesses via indirect calls, enforced by a runtime mechanism that inspects the full call chain on the current stack.

While such an access control policy is simple enough to be used in practice to protect users and systems from many hostile programs downloaded over a network, it does not restrict information flows, which may inadvertently allow unauthorized code to influence or be influenced by the execution of security-sensitive code. This limitation can lead to subtle security vulnerabilities that programmers find difficult to identify. Previous proposals attempt

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Acces; Article No. 23; pp. 23:1–23:34

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to strengthen this security model by exploring variants of the operational semantics that yield stronger theorems, for example, using a stricter call relation that potentially prohibits many good programs [1] or applying a transitive integrity policy without considering confidentiality [24]. In addition, these security models are not enforced statically by the compiler, and may rely on programmer disciplines to provide appropriate runtime checks.

In this section, we review the access control model in Java/CLR and examine its security requirements and limitations. Then we informally introduce our solution to this problem—a general security model that provides an intransitive information flow policy using the access control specification of Java/CLR.

1.1 Access Control Model in Java/CLR

The access control model used in languages, such as Java and CLR, guards access to system resources [14]. It specifies which privileges, in the form of permissions, are allowed for code from specific sources. Before invoking security-sensitive code which operates on system resources or sensitive information, a security check is performed against the execution stack to verify that all callers currently on the stack have been granted sufficient privilege. When the inspection fails, i.e., some caller on the stack is not authorized, a security exception is thrown. Such a runtime security check is often referred to as *stack inspection*. The purpose of stack inspection is to provide an intransitive access control policy to prevent *confused deputy* attacks [17], when unauthorized code accesses security-sensitive operations indirectly (transitively) by calling authorized code.

Stack inspection is often used with the intent of preventing unwanted information flow between untrusted code and security-sensitive code [18]. Typically a security check is performed before confidential information (produced by security-sensitive code) is released, or tainted information (produced by untrusted code) is used. However, such an access control mechanism cannot prevent information-based attacks, because stack inspection is insufficient to track information propagation. Confidential information may be leaked to unauthorized code after stack inspection—a confidentiality violation. As well, tainted data left by unauthorized code that has exited the execution stack before stack inspection may be used in security-sensitive operations—an integrity violation. Patterns abstracted from real-world programs are used to describe some of these issues, and are illustrated in Figures 1–3. The examples use Java syntax and its security check, but they are also applicable to CLR.

Figure 1 illustrates the example of a resource that needs to be protected from unauthorized access, as often found in libraries. The private native method `L.create` is security sensitive, which creates a system resource with the specified name. It may be accessed via the public method `L.createResource`, and the created resource may be stored in a private field and returned by the public method `L.getResource`. In order to prevent unwanted interference from unauthorized application code, a pair of security checks are enforced programmatically to guard the creation and release of the resource in methods `L.createResource` and `L.getResource` respectively via the permission `Permission("resource")`. The method `M.defaultResource` creates a default resource via `L.createResource` using the default resource name. The design intent of these classes is that the resource can be created and used by only an authorized caller that has been granted `Permission("resource")`.

Unfortunately, this design intent can be easily violated, as illustrated in Figure 2. The permissions granted to all classes are summarized in Table 1. The class `L` containing system operations may be part of the library that is typically trusted and thus granted full privilege (via the special `AllPermission`). The class `M` that uses `L` is also part of the full trusted library.

```
1  public class L {
2      private Resource resource;
3      private native Resource create(String name);
4      public void createResource(String name) {
5          AccessController.checkPermission(
6                  new Permission("resource"));
7              resource = create(name);
8      }
9      public Resource getResource() {
10         AccessController.checkPermission(
11                 new Permission("resource"));
12         return resource;
13     }
14 }
15
16 public class M {
17     L l = ...;
18     public void defaultResource() {
19         String name = "default";
20         l.createResource(name);
21     }
22 }
```

Figure 1 A resource is guarded by permission checks.

Class	Permissions
L	{AllPermission}
M	{AllPermission}
A	{Permission("resource")}
B	∅
C	∅

Table 1 Granted permissions.

The class A is not fully trusted, as a typical application using the library, but it is granted the resource permission. The class B is not trusted, thus should never create nor use any resource. However, B can do both things indirectly via the authorized class A.

The stack inspection performed by security checks can find only the currently active callers; therefore it cannot inspect callers that have exited the stack before the check or will enter the stack after the check. In Figure 2, the tainted string "secret" from the class B may be used by the authorized class A to create a resource with that name. This is because, when L.createResource is called (Line 6 Figure 2) in A , the unauthorized call b.make has already exited the stack, so the security check in L.createResource sees only the authorized caller A.main and grants access. Similarly, the confidential resource created in L.create may be undesirably leaked and used by the unauthorized class B. This is because, when L.getResource is called (Line 8 Figure 2) in A, the security check in L.createResource sees only the authorized caller A.main and grants access. After the security check, the resource may flow to unauthorized code without security check as shown on Line 9 in Figure 2.

Thus the stack inspection model cannot detect unwanted information propagation. Furthermore, the model is overly restrictive, and prevents certain operations that are secure. In Figure 3, the unauthorized class C calls on M.defaultResource, which would throw a security exception because L.createResource has a security check. However, this code is safe because the default resource is created with a library-provided string, regardless of user

```
1  public class A {
2      public static void main(String[] args){
3              L l = ···;
4              B b = ···;
5              String name = b.make();
6              l.createResource(name);
7              ···
8              Resource r = l.getResource();
9              b.use(r);
10     }
11 }
12
13 public class B {
14     public String make() {
15         return "secret";
16     }
17     public void use(Resource res) { ··· }
18 }
19
```

Figure 2 Unauthorized code may send/receive information to/from security-sensitive operation.

```
1  public class C {
2      public static void main(String[] args) {
3          M m = ···;
4          m.defaultResource();
5      }
6  }
```

Figure 3 A secure program that is rejected.

input (unauthorized code cannot influence the choice of resource to be created). To get around this restriction, library programmers can place the calls to `L.createResource` inside a privileged block, which is a special code section used for privilege escalation. In general, privileged blocks should be avoided as much as practical, because they break the usual access control and may lead to security vulnerabilities [18].

These examples illustrate the difficulty of specifying the security intent for a general class of interacting trusted and untrusted code. Library code is often privileged to communicate with both untrusted code and security-sensitive code, but access control needs to be carefully designed to disallow unwanted information flow between application code and security-sensitive code. Relying solely on the programmer could cause errors that are potentially exploitable [18].

1.2 Secure Information Flow by Access Control

Informally, Java/CLR's stack-based access control model aims to prevent untrusted code from causing harm. However, it is surprisingly hard to state a useful theorem that captures this intent for a general class of trusted and untrusted code [12]. We approach this problem by identifying the functional security requirements of applications relying on such an access control mechanism and then developing a security model to capture these requirements.

First, as seen in the example, the access control mechanism is often used with the intent of enforcing confidentiality and integrity policies to prevent unwanted information flow between untrusted and security-sensitive code. Second, users, rather than developers, provide finer-grained security control over the components of programs to determine their

capabilities to propagate information, by explicit authorization in the form of permission. Third, information flow should be controlled by an intransitive policy to avoid unwanted transitive information propagation between program components. In the example, information flow is not allowed between the application code and security-sensitive code even though they can both communicate with the library code. Fourth, it is desirable to allow mutable or cyclic information flow between code with different privileges. For example, untrusted code granted a specific permission may send and receive information from security-sensitive code that requires the permission. Finally, the stack-based access control mechanism relies on programmer disciplines to provide necessary access control checks to protect sensitive operations, which can be both error-prone and excessive. It is advantageous for the security guarantee to be automatically provided by the compiler.

Previous attempts explore variants that yield stronger security than stack inspection. History-based access control [1] provides access control based on code previously executed (and not just the code currently on the stack). It may prevent authorized code from executing a security-sensitive operation if less trusted code was previously executed, and does not prevent information flow to unauthorized code that has not been previously executed. Information-based access control [24] uses a lattice-based information flow system to enforce its integrity policy. In addition to the lack of confidentiality, it supports neither intransitive information flow nor cyclic information flow between code at different security levels.

In lattice-based multilevel security systems, security levels form a lattice and each variable may be associated with a security level. To ensure confidentiality, information flow from higher-level to lower-level variables should not be allowed. On the other hand, to ensure integrity, flows to higher-level variables should be restricted. Therefore, having both integrity and confidentiality in the same program would preclude information flows between different security levels. In our example, to allow information flows between the class `A` and class `L` would require both classes have the same privileged. This is undesirable, because it would either grant the class `A` with unnecessary privilege or reduce the privilege of the class `L` to use other privileged code (e.g. other parts of the library). Cross product of lattices (i.e. combining integrity and confidentiality labels) handle both integrity and confidentiality; however they do not provide intransitive policies.

We tackle the problem by introducing a new security model that reuses Java/CLR's existing access control specification and extends it with explicit security requirements using Dual-Access labels (DAL) which is formally introduced in Section 2. It provides a simple and more general model that captures intransitivity of access control in an information flow security model. In this model, each piece of program code is assigned a pair of security levels, based on the observation that the security requirements on the code can be specified in terms of both *accessibility* (the privilege *required* to access the code) and *capability* (the privilege *granted* to the code). The level of trust and secrecy of code is determined by the security levels representing capability and accessibility respectively—holding more permissions means more capable (i.e., more trustworthy), and requiring more permissions means less accessible (i.e., has more secrets).

DAL can encode untrusted code, trusted code and security-sensitive code in Java/CLR, which are special cases of our security model, using the different access control specifications. Untrusted code requires no permission (anyone can influence it) and holds permissions that are granted explicitly via the policy file (it can influence code that demands only fewer permissions). Trusted code, typically libraries, on the other hand, hold all permissions (trusted to influence any other code). Public methods in trusted code require no permission, because they are accessible by anyone, while security-sensitive code may require specific

Code	Granted Permissions	Required Permissions
`L.create`	{`AllPermission`}	{`Permission("resource")`}
`L`	{`AllPermission`}	∅
`M`	{`AllPermission`}	∅
`A`	{`Permission("resource")`}	∅
`B`	∅	∅
`C`	∅	∅

Table 2 Granted and required permissions.

permissions because its accessibility must be held by other code in order to influence it.

Informally, our security policy requires both the source and sink of the information propagation to respect each other's DAL, providing both integrity and confidentiality guarantees between code with different pairs of security levels. To ensure confidentiality, the capability of the sink must have sufficient privilege to meet the accessibility of the source. To ensure integrity, the capability of the source must have sufficient privilege to meet the accessibility of the sink.

Table 2 extends the granted permissions shown in Table 1 with a set of required permissions in order to identify the security violations as well as to permit the execution of the safe program in the examples described in Section 1.1. All code, except `L.create`, has no particular requirement for how it is accessed. The method `L.create` is security sensitive because it requires the specific resource permission `Permission("resource")` to be held by anyone who can send/receive information to/from it. Like permissions in Java/CLR, DAL may be specified in a security policy file for either classes or methods. All entities within the same class/method will have the same DAL. Method-level DAL is more specialized and will override the DAL on the enclosing class. DAL cannot be inherited by subclasses or overriding methods.

By information flow analysis, the string `"secret"` in the class `B` may flow to the formal parameter of the sensitive method `L.create`. By applying our access control policy, which requires that the capability of source `"secret"` (an empty permission set) must be greater than the accessibility of the sink formal parameter of `L.create` (a set containing the resource permission), we can conclude that there is an insecure information flow. Similarly, the returned value of the sensitive method `L.create` may flow to the untrusted class on the formal parameter of `B.use`. This violation can also be identified by using our access policy where the capability of the sink `B` (an empty permission set) is required to be greater than the accessibility of the source `L.create` (a set containing the resource permission). On the other hand, the class `C` can meet the security policy. The string `"default"` is defined in the trusted class `M` with full privilege, therefore our access control policy would allow it to flow to `L.create`.

Often, finer-grained access control is required to meet more-precise security requirements. For example, different permissions may be used to guard the creation and release of the resource in Figure 1. The permission checked at Line 6 may be replaced by a more specific permission `Permission("createResource")`, and the permission at Line 11 may be also replaced by a more specific permission `Permission("getResource")`. Such precision would require a finer-grained access control policy that can be specified in our model as shown in Section 2.1.

1.3 Contributions

In this paper, we introduce DAL, a new security model that extends the access control model of Java/CLR to control information flow between code with different security requirements. A DAL-based security policy does not need be transitive as is often assumed in existing work; thus it is strictly more expressive than lattice-based transitive security models. It provides more flexible information flow relations than traditional lattice-based security models, allowing permitted information flows across different security labels and disallowing undesired interactions without demanding transitive information flows. The separation of required privileges (specified by code developer) and granted privileges (specified by the user of the code according to its source; note that the developer cannot grant privileges) enables to cleanly specify and check all security requirements of the example as identified in the second paragraph of Section 1.2. To the best of our knowledge, no existing techniques are able to do that.

To enforce possibly intransitive security policies statically at compile-time, we present a generic security type system that tracks both explicit and implicit information flows, raised from both conditionals and virtual dispatches in object-oriented programs. By using a points-to analysis, it propagates security labels along with information flow and ensures that specified security label on each variable is respected by all possible values flow to it, therefore supporting intransitive security policies like DAL as well as traditional transitive security policies. We provide a big-step operational semantics and prove the subject reduction theorem.

Informally, the type system verifies that, for any operation (e.g. security-sensitive code or untrusted code), all the code responsible for (i.e. may transitively influence) the operation is sufficiently authorized by the given security policy. Formally, the type system provides a new notion of intransitive noninterference that handles confidentiality and integrity in a unified framework. It generalizes the standard notion of noninterference in lattice-based security models, because it does not require transitivity. We prove the intransitive noninterference theorem with a generalized definition of indistinguishability based on values, variables, and history associated with information flows.

The rest of the article is organized as follows. Section 2 formalizes DAL and its security policy. Section 3 describes the security type system that enforces the security model statically. Section 4 provides a big-step operational semantics and the subject reduction theorem. Section 5 presents the main result—an intransitive noninterference property for security policies that need not to be transitive. Section 6 discusses background and related work. Section 7 concludes the paper. Additional proofs for the properties in Section 4 and 5 are available as supplementary material.

2 Dual-Access Labels

In this section, we develop the formal details of DAL, including the security policy and some basic properties, before we show how to enforce it statically by a type system in Section 3.

In multilevel security [29], each program variable is assigned a security level. The security levels form a lattice, partially ordered by $\leq$ with the top and bottom elements denoted by $\top$ and $\bot$ respectively. For information flow security, a security (confidentiality or integrity) policy, is expressed in terms of the lattice.

In our new security model, each program variable is labeled with a corresponding DAL (represented by φ) that defines the level of access control required for the variable. DAL is formed by a pair of security levels in a lattice-based multilevel security model.

$$\varphi \quad ::= \quad \mathcal{A} \cdot \mathcal{C}$$

The security levels $\mathcal{A}$ and $\mathcal{C}$ determine respectively the accessibility and capability of the variable for transferring information between variables. Together they restrict how information propagates in the system by the security policy defined by the relation $\rhd$ defined by [ACCESS]. Intuitively, information in variables with higher-level accessibility and lower-level capability is more restricted than information in variables with lower-level accessibility and higher-level capability.

$$\text{[ACCESS]} \quad \frac{\mathcal{A}_1 \leq \mathcal{C}_2 \qquad \mathcal{A}_2 \leq \mathcal{C}_1}{\mathcal{A}_1 \cdot \mathcal{C}_1 \ \rhd \ \mathcal{A}_2 \cdot \mathcal{C}_2}$$

The term *Dual-Access* refers to the concept of a two-way obligation that both ends of information flow are obliged to meet the security requirement of each other. That is, the sender must be sufficiently privileged to send information to the receiver, as well as the receiver must be sufficiently privileged to receive information from the sender. In the above rule, $\mathcal{A}_1 \cdot \mathcal{C}_1$ is the DAL of the source of the information while $\mathcal{A}_2 \cdot \mathcal{C}_2$ is the DAL of the receiver. Both the sender and receiver must be authorized to exchange information, providing both confidentiality and integrity. To ensure confidentiality, the receiver is allowed to receive information from the sender only if it has sufficient capability to satisfy the sender's accessibility (the security level representing the capability of the receiver $\mathcal{C}_2$ is greater than or equal to the security level representing the accessibility of the sender $\mathcal{A}_1$). To ensure integrity, the sender is allowed to send information to the receiver only if it has sufficient capability to satisfy the receiver's accessibility (the security level representing the capability of the sender $\mathcal{C}_1$ is greater than or equal to the security level representing the accessibility of the receiver $\mathcal{A}_2$).

In general, the security policy [ACCESS] is neither transitive nor reflexive. It is reflexive only when the capability of DALs is alway greater than or equal to the accessibility (i.e., $\mathcal{A} \leq \mathcal{C}$ for all DALs). The security policy is transitive only when the accessibility of DALs is alway greater than or equal to the capability (i.e., $\mathcal{C} \leq \mathcal{A}$ for all DALs). Traditional lattice-based security policies are a special case in our security model when the accessibility of a variable is always the same as its capability (i.e., $\mathcal{A} = \mathcal{C}$ for all DALs, meaning accessibility and capability of variables are not distinguished), implying both transitivity and reflexivity. However, since reflexivity is often needed in practice (e.g. assignments between variables with the same label should be allowed), we will generally use the relation $\trianglerighteq$ (i.e., the reflexive closure of $\rhd$) to constraint information flow.

Although the DALs do not form a lattice under $\rhd$, intuitively, accessibility of a higher security level implies more secret code, thus restricting access to its variables, while accessibility of lower security level implies less secret code. On the other hand, capability of higher security level implies more privilege, enabling the code to access other code. Therefore, DALs can be ordered by a transitive, reflexive subtyping relation $\sqsubseteq$, which is not used to govern assignment but by allowing covariance on accessibility and contravariance on capability.

$$\text{[SUBTYPING]} \quad \frac{\mathcal{A}_1 \leq \mathcal{A}_2 \qquad \mathcal{C}_2 \leq \mathcal{C}_1}{\mathcal{A}_1 \cdot \mathcal{C}_1 \ \sqsubseteq \ \mathcal{A}_2 \cdot \mathcal{C}_2}$$

This subtyping relation preserves the access control policy and is used in the subsumption rules in the static semantics (see Table 5).

▶ Lemma 1 (Subtyping Preserves Access). For any labels φ_1, φ_2, φ_3 and φ_4:

$$\left.\begin{array}{l}\varphi_1 \trianglerighteq \varphi_2\\ \varphi_3 \sqsubseteq \varphi_1\\ \varphi_4 \sqsubseteq \varphi_2\end{array}\right\} \Longrightarrow \varphi_3 \trianglerighteq \varphi_4$$

Proof. The proof is straightforward by [SUBTYPING] and [ACCESS].

The union operation $\sqcup$ combines two DALs by the join (least upper bound) of the security levels of their accessibility and the meet (greatest lower bound) of the security levels of their capability. Note that, this union operation is the join of DALs with regard to the ordering $\sqsubseteq$.

[UNION]
$$\mathcal{A}_1 \cdot \mathcal{C}_1 \sqcup \mathcal{A}_2 \cdot \mathcal{C}_2 = \mathcal{A}_1 \vee \mathcal{A}_2 \cdot \mathcal{C}_1 \wedge \mathcal{C}_2$$

As all information transfers are governed by a potentially intransitive policy the history of information transfer must be tracked. This is done by taking the union of labels of variables the information has flowed through. For example, assume we have two independent secure assignments $y = x$ and $z = y$ (implying the DAL of x can access the DAL of y, we can write $label(x) \rhd label(y)$, and similarly $label(y) \rhd label(z)$). Their composition $y = x; z = y$ may not necessarily be secure because information can flow from x to z, requiring $label(x) \rhd label(z)$ that is not derivable from the given assumptions (because $\rhd$ may not be transitive). Our security model identifies this violation by propagating labels along with information flows. After the assignment $y = x$, the label of the value stored in y captures $label(x)$. When $z = y$ is evaluated, the label of the value read from y would be the union of both labels: $label(x) \sqcup label(y)$. Then the access control policy enforces $(label(x) \sqcup label(y)) \rhd label(z)$.

2.1 Extension for Antisymmetric Security Policy

The security policy we have described so far ([ACCESS]) is symmetric, which is useful in many applications where authorized code needs to read from and write to security-sensitive data/code. However, in practice there are also cases where applications should only be allowed to either read from or write to certain security-sensitive data. For instance, in Java, reading and writing system properties may require different permissions; applications with privileges to read system properties may not be allowed to write them. An antisymmetric security policy is required to express such security requirements. Antisymmetric security policy can be obtained by using an extended Dual-Access Label or xDAL for short and represented by

$$\varphi \quad ::= \quad \mathcal{I} \cdot \mathcal{F} \cdot \mathcal{C}$$

In DAL, a single security level is used to represent the accessibility of a variable, controlling both delivery and reception of information from/to the variable. In xDAL, accessibility ($\mathcal{A}$) is split into two parts: $\mathcal{I}$ sets up the integral security requirement and controls reception (write) of information, while $\mathcal{F}$ sets up confidential requirements and controls delivery (read) of information. The security policy can be refined by using xDAL.

[EXT-ACCESS]
$$\frac{\mathcal{F}_1 \leq \mathcal{C}_2 \qquad \mathcal{I}_2 \leq \mathcal{C}_1}{\mathcal{I}_1 \cdot \mathcal{F}_1 \cdot \mathcal{C}_1 \ \rhd \ \mathcal{I}_2 \cdot \mathcal{F}_2 \cdot \mathcal{C}_2}$$

This security policy is antisymmetric and restricts only the confidential accessibility of the sender and the integral accessibility of the receiver. It is strictly more flexible than the

security policy for DAL because it removes the requirements on confidentiality of the receiver and integrity of sender.

The subtyping relation for xDAL is unsurprising, where both confidential accessibility and integral accessibility are covariant.

[EXT-SUBTYPING]
$$\frac{\mathcal{I}_1 \leq \mathcal{I}_2 \qquad \mathcal{F}_1 \leq \mathcal{F}_2 \qquad \mathcal{C}_2 \leq \mathcal{C}_1}{\mathcal{I}_1 \cdot \mathcal{F}_1 \cdot \mathcal{C}_1 \ \sqsubseteq \ \mathcal{I}_2 \cdot \mathcal{F}_2 \cdot \mathcal{C}_2}$$

Similarly, two xDALs can be combined by the joins of their confidential accessibility and integral accessibility, respectively.

[EXT-UNION]
$$\mathcal{I}_1 \cdot \mathcal{F}_1 \cdot \mathcal{C}_1 \sqcup \mathcal{I}_2 \cdot \mathcal{F}_2 \cdot \mathcal{C}_2 = \mathcal{I}_1 \vee \mathcal{I}_2 \cdot \mathcal{F}_1 \vee \mathcal{F}_2 \cdot \mathcal{C}_1 \wedge \mathcal{C}_2$$

3 Type System

In this section, we show how our security model of DAL can be analyzed statically. We present a generic security type system to enforce any information flow policy on security labels that can provide $\rhd$, $\sqsubseteq$, $\sqcup$ relations as defined in Section 2. Therefore, it can support intransitive security policies (where $\rhd$ provides an intransitive relation) like DAL and xDAL, as well as traditional lattice-based transitive security policies (where $\rhd$ provides a transitive relation). The main result of the type system is a new notion of intransitive noninterference property, to be discussed in Section 5.

To formalize the type system, we choose a core language similar to [12, 24], but with the addition of class inheritance and dynamic dispatch to more closely model conventional object-oriented languages. Our security model restricts how information is transferred among program variables, and does not place any restriction on the calling chain (though such access control can also be enforced statically using the same security labels). Our language does not have runtime access control such as permission checks, since it does not add to the information flow security properties being developed.

Security type systems for enforcing transitive security policies have enjoyed simpler type checking, by exploiting the transitive type relations. Security labels are associated with data types and are checked in the same way as data types. To support intransitive security policies, in addition to type checking, we also need to track the full history of information flow by propagating security labels along assignments. This propagation of security labels is similar to the propagation of points-to sets in points-to analysis .

Points-to analysis is a static program analysis technique to approximate the set of objects (abstracted by their allocation sites) that may be pointed or referenced by program variables [30]. Points-to sets are computed by propagating points-to information along assignments until reaching a stable state (fixed-point), using a fixed-point algorithm. Based on this observation, we associate our security labels with points-to sets and propagate them together with points-to sets, thus our analysis can be easily implemented in a standard points-to analysis. Moreover, it is well known that the use of points-to information can substantially improve the accuracy of static analyses. We formulate our type system as an inter-procedural, field-sensitive points-to analysis (thus both the accuracy and efficiency are determined by the points-to analysis).

3.1 Syntax

The syntax of the core language is given in Figure 4. For simplicity, expressions are assigned to local variables before use (like three-address code). Class types are standard thus are not explicitly represented in our language and typing rules; we assume class type soundness (memory safety) of input programs. We assume methods are not overloaded (i.e., they are distinguished only by their names not by their signatures).

$$
\begin{array}{lcl}
C & ::= & \mathsf{class}\ c\ [\mathsf{extends}\ c]\ \{\overline{f};\ \overline{M}\} \\
M & ::= & m(x)\ \{s\} \\
s & ::= & x{=}\mathsf{new}^{o}\ c \mid x{=}e \mid x.f{=}x \\
 & & \mid x.m(x) \mid \mathsf{if}\ x\ \mathsf{then}\ s\ \mathsf{else}\ s \mid s;s \\
e & ::= & x \mid x.f
\end{array}
$$

Figure 4 Abstract syntax for the core language.

Class, field, variable names are unique in their respective defining scopes. The special variable this is a read-only variable that refers to the receiver object in method bodies. For computing points-to sets, object creation is labeled with its allocation site (i.e. the program counter of the new object instruction). Note that, the if test branches on null test value.

Security requirements are typically specified in separate policy files, so we do not explicitly include labels in the language syntax. Instead, a simple function is used to look up the specified security labels for variables and fields, e.g. $\mathit{label}(x)$ or $\mathit{label}(f)$. We assume fully qualified field/variable names (i.e., by defining class and method names) are used for lookup. In the case of DAL/xDAL where security labels may be specified on code (either class or method), $\mathit{label}(x)$ refers to the security label on the method that defines variable x while $\mathit{label}(f)$ refers to the security label on the class that defines field f. New objects are always assigned to a local variable before use, therefore they do not need labels.

3.2 Static Semantics

The type system enforces security policies by propagating security labels along with information flows between program variables, and ensuring they cannot violate the specified security labels on variables. In order to deal with intransitive security policies, the history of information propagation is tracked by union of labels of variables the information has flown through.

We formalize the static semantics of our type system in Figure 5 as a set of inference rules. The expression typing is defined by

$$A \vdash e : \tau\ \varphi$$

where τ is the points-to set of the expression e (the set of allocation sites which the expression e may be evaluated to at runtime), and φ is the security label of information that may *explicitly* propagate to e via reading variables/fields. The statement typing is defined by

$$A \vdash s : \varphi$$

where φ tracks the security label of information that may *implicitly* influence the execution of the statement s through the program control flow (in our language, it is the label of either the conditional variable or the base variable of an invocation).

Explicit information flow is tracked by propagating labels between variables and fields via direct assignments, while implicit information flow is tracked by propagating labels from

$$\frac{\Gamma(x) = \tau\ \varphi}{\Gamma\ \Sigma \vdash x : \tau\ \ \varphi}\ [\text{VAR}] \qquad \frac{\Gamma(x) = \tau_0\ \varphi_0 \qquad \forall o \in \tau_0\ \cdot \begin{cases} \Sigma(o)(f) = \tau_1\ \varphi_1 \\ \tau_1 \subseteq \tau \qquad \varphi_1 \sqsubseteq \varphi \end{cases}}{\Gamma\ \Sigma \vdash x.f : \tau\ \ \varphi \sqcup \varphi_0}\ [\text{LOAD}]$$

$$\frac{\Gamma(x) = \tau\ \varphi \qquad \Gamma\ \Sigma \vdash e : \tau\ \varphi}{\Gamma\ \Sigma \vdash x{=}e : \varphi}\ [\text{ASSIGN}] \qquad \frac{\Gamma(x) = \tau\ \varphi \qquad o \in \tau}{\Gamma\ \Sigma \vdash x{=}\mathsf{new}^o\ c : \varphi}\ [\text{NEW}]$$

$$\frac{\Gamma(x) = \tau_0\ \varphi_0 \qquad \Gamma(y) = \tau_1\ \varphi_1 \qquad \forall o \in \tau_0\ \cdot \begin{cases} \Sigma(o)(f) = \tau_2\ \varphi_2 \\ \tau_1 \subseteq \tau_2 \qquad \varphi \sqcup \varphi_0 \sqcup \varphi_1 \sqsubseteq \varphi_2 \end{cases}}{\Gamma\ \Sigma \vdash x.f{=}y : \varphi}\ [\text{STORE}]$$

$$\frac{\begin{array}{c}\Gamma(x) = \tau_0\ \varphi_0 \qquad \Gamma\ \Sigma \vdash y : \tau_1\ \varphi_1 \\ \forall o \in \tau_0\ \cdot \begin{cases} method(type(o), m) = m(z)\{s\} \\ \{\mathsf{this} \mapsto (\tau_0\ \varphi_0), z \mapsto (\tau_1\ \varphi_1)\}\ \Sigma \vdash s : \varphi \sqcup \varphi_0 \end{cases}\end{array}}{\Gamma\ \Sigma \vdash x.m(y) : \varphi}\ [\text{CALL}]$$

$$\frac{A \vdash x : \tau_0\ \varphi_0 \qquad A \vdash s_1 : \varphi \sqcup \varphi_0 \qquad A \vdash s_2 : \varphi \sqcup \varphi_0}{A \vdash \mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 : \varphi}\ [\text{IF}] \qquad \frac{A \vdash s_1 : \varphi \qquad A \vdash s_2 : \varphi}{A \vdash s_1; s_2 : \varphi}\ [\text{SEQ}]$$

$$\frac{A \vdash e : \tau_1\ \varphi_1 \qquad \tau_1 \subseteq \tau \qquad \varphi_1 \sqsubseteq \varphi}{A \vdash e : \tau\ \varphi}\ [\text{SUB-EXP}] \qquad \frac{A \vdash s : \varphi_1 \qquad \varphi \sqsubseteq \varphi_1}{A \vdash s : \varphi}\ [\text{SUB-STM}]$$

Figure 5 Static semantics.

parent statements to their sub-statements (i.e. branches or method definitions selected by dynamic dispatch). Two subsumption rules [SUB-EXP] and [SUB-STM] are provided for expressions and statements respectively. For expressions, it is conservative to enlarge the points-to set and label of explicit flows because expressions are read-only (implying our points-to analysis is subset-based like [30]).

On the other hand, as for statements, it is conservative to reduce the label of implicit flows because statements are write-only (executed for only side effects). For example, let us consider an assignment statement $x{=}y$ typed with an implicit flow label φ. Because the update on x is influenced by the implicit flow (in addition to the explicit flow from y), the security label of x must be able to capture such implicit flow by requiring $\varphi \sqsubseteq \mathit{label}(x)$. Therefore, φ may be substituted with its subtype without altering this requirement.

All expressions and statements are typed in the environment A, an abstract state to approximate runtime states:

$$A \quad ::= \quad \Gamma\ \ \Sigma$$

The abstract stack Γ tracks local typing environment on the current call frame, mapping local variables to labeled points-to sets—a pair of points-to set and label. For example, $\Gamma(x) = \tau\ \varphi$ means the variable x may contain an object created at one of the allocation sites in τ and has the security label φ which is the union of all security labels of information may propagate to x.

The abstract heap Σ tracks global typing environment on the shared heap, mapping allocation sites and fields to labeled points-to sets, e.g. $\Sigma(o)(f) = \tau\ \varphi$. Field-sensitivity of analysis is enabled by distinguishing the points-to sets of each field (see [LOAD] and [STORE]

rules). Γ and Σ together represent the *points-to graph* used in points-to analyses. Although our type system is flow-insensitive, it can be used to analyze static single assignment(SSA)-based intermediate representations where flow-sensitivity of data flow is implicitly provided.

Like a points-to analysis, our static semantics define constrains on points-to sets and security labels which are to be solved by a fixed-point computation [30]. The typing rules do not explicitly enforce a security policy, instead they track information propagation in an abstract state which is required to be well-formed. The [ABSTRACT STATE] rule describes the conditions satisfied by a well-formed abstract state where each variable/field is well-typed by its security label according to a given security policy.

$$\frac{\begin{array}{c}\forall x \quad \cdot \quad \Gamma(x) = \tau\ \varphi \implies \varphi \trianglerighteq \mathit{label}(x) \quad \mathit{label}(x) \sqsubseteq \varphi \\ \forall o,f \quad \cdot \quad \Sigma(o)(f) = \tau\ \varphi \implies \varphi \trianglerighteq \mathit{label}(f) \quad \mathit{label}(f) \sqsubseteq \varphi\end{array}}{\vdash \Gamma\ \Sigma} \text{[ABSTRACT STATE]}$$

The security label of the value in the variable/field must satisfy (be able to access) the specified security label of the variable/field for a given security policy. For example, the DAL-specific security policy is enforced by constraints on the DAL labels that any abstract state needs to satisfy in order to be well-formed. Any runtime state, to be discussed in Section 4, of a well-typed program will also be well-formed.

The expression rules [VAR] and [LOAD], in addition to looking up the labeled points-to set from the abstract stack and heap respectively, track explicit information flow by including the security label of the source variable/field (where the value is read) in the combined label of the expression. Since field selection depends on the value of the receiver, its label also includes the label of the receiver.

The assignment rules [ASSIGN], [STORE] and [NEW] ensure the points-to set and label of assigned expressions are captured in the points-to set and label of the respective variable/field in the abstract state. In addition, the label of implicit flow of the assignments must also be subsumed by the label of respective variable/field in the abstract state (by [SUB-STM]). In [NEW], the allocation site of the new object is captured and the label of new object is not significant (e.g. may be considered as $\bot \cdot \top$ in DAL).

The [CALL] rule looks up all possible method targets based on the points-to set of the receiver variable x and analyses the method definitions in the new abstract stack constructed from the labeled points-to sets of the actual method parameters. Note that we use the expression typing rule on actual method parameters, instead of looking them up from the original stack, to allow them to be covered by the subsumption rule.

For notational simplicity, $\mathit{type}(o)$ represents the class type of the object created at the allocation site o. We assume a standard method lookup definition (indicated by $\mathit{method}(c, m)$) which searches for method definition, starting from the given class to its super classes.

$$\frac{\mathsf{class}\ c\ \ldots\ \{\ldots m(x)\ \{s\}\ldots\}}{\mathit{method}(c,m) = m(x)\ \{s\}} \text{[METHOD-DEFINE]}$$

$$\frac{\mathsf{class}\ c\ \mathsf{extends}\ c_0\ \{\overline{f};\ \overline{M}\} \qquad (m(\_)\ \{\_\}) \notin \overline{M}}{\mathit{method}(c,m) = \mathit{method}(c_0,m)} \text{[METHOD-INHERIT]}$$

Note that our type analysis is context-sensitive, because each method invocation is analyzed independently in a new abstract stack. Cheaper context-insensitive analysis can also be supported by simply using a shared abstract stack for all invocations, effectively merging all points-to sets and security labels of the same local variable used in different method invocations.

Since dynamic dispatches allow implicit information flow through the selection of methods based on the class type of the receiver object, [CALL] propagates the combined label of the

parent statement φ and the label of the receiver object φ_0 to the typing of the body of all methods that may be selected at runtime. Similarly, in addition to propagating the security label of implicit flows from parent statements, [IF] also include the label of the conditional variable to the analysis of both branches.

4 Dynamic Semantics

This section presents dynamic semantics and subject reduction theorem of the type system to show the points-to information and security labels are preserved over evaluation. The security property of the type system, the intransitive noninterference theorem, is to be discussed in detail in Section 5.

The security labels are not needed at runtime, since security policies are enforced statically by the type system. However, in order to prove the soundness and security properties of the type system, we need to connect static semantics to a form of dynamic semantics that retains security labels at runtime. The dynamic semantics is formalized using the big-step style operational semantics, where labels are tracked during evaluation. Like static semantics, explicit information flow is tracked by propagating labels between variables and fields via direct assignments, while implicit information flow is tracked by propagating labels from parent statements to their sub-statements.

4.1 Operational Semantics

The big-step operational semantics for the core language is given in Figure 6, using additional notation to represent runtime values and states:

$$\begin{array}{lcl} e & ::= & \ldots \mid v \\ v & ::= & l^o \mid \mathsf{null} \\ E & ::= & S\ H \end{array}$$

The value v is the result of evaluating an expression, may be either null or a heap location l labeled with its allocation site o (omitted when not used). The execution state E consists of a concrete stack S and a concrete heap H. The stack S maps local variables to labeled values—a pair of value and label. For example, $S(x) = v\ \varphi$ means the variable x contains the value v and its label φ (combined from all variables/fields that may have influenced the value). Similarly, the heap H maps locations and fields to labeled values, e.g. $H(l)(f) = v\ \varphi$.

Statements are evaluated for their side effects on input state in the form of

$$s\ \varphi\ E_1\ \Downarrow\ E_2$$

The label φ tracks implicit information flow that may influence the evaluation of statement s from the input state E_1 to the output state E_2 (see [E-TRUE/FALSE] and [E-CALL]). A program is a statement evaluated in the initial configuration $s\ \varphi\ \emptyset\ \emptyset$ where s is the body of the main method, both the stack and heap are empty, and φ is the initial label with least restriction (e.g. $\bot{\cdot}\top$ for DAL and $\bot{\cdot}\bot{\cdot}\top$ for xDAL).

Expressions are evaluated to labeled values in the form of

$$e\ E\ \Downarrow\ v\ \varphi$$

This evaluation has no side effects, and is not affected by implicit information flows and does not change the input state.

Expression evaluations [E-VAR] and [E-LOAD] are similar to their static counterparts, looking up the labeled values from the stack and heap respectively and tracking explicit information

$$\text{[E-VAR]}\ \frac{S(x) = v\ \varphi}{x\ S\ H\ \Downarrow\ v\ \varphi} \qquad \text{[E-LOAD]}\ \frac{S(x) = l\ \varphi_0 \qquad H(l)(f) = v\ \varphi}{x.f\ S\ H\ \Downarrow\ v\ \varphi \sqcup \varphi_0}$$

$$\text{[E-ASSIGN]}\ \frac{e\ S\ H\ \Downarrow\ v\ \varphi_0}{x{=}e\ \varphi\ S\ H\ \Downarrow\ S[x \mapsto (v\ \varphi \sqcup \varphi_0 \sqcup label(x))]\ H}$$

$$\text{[E-NEW]}\ \frac{x \notin dom(S) \qquad l^o \notin dom(H) \qquad fields(c) = \overline{f}}{(x{=}\mathsf{new}^o\ c)\ \varphi\ S\ H\ \Downarrow\ S, x \mapsto (l^o\ \varphi \sqcup label(x))\ H, l^o \mapsto (\overline{f \mapsto (\mathsf{null}\ label(f))})}$$

$$\text{[E-STORE]}\ \frac{S(x) = l\ \varphi_0 \qquad y\ S\ H\ \Downarrow\ v\ \varphi_1}{x.f{=}y\ \varphi\ S\ H\ \Downarrow\ S\ H[l \mapsto H(l)[f \mapsto (v\ \varphi \sqcup \varphi_0 \sqcup \varphi_1 \sqcup label(f))]]}$$

$$\text{[E-CALL]}\ \frac{\begin{array}{c} x\ S\ H\ \Downarrow\ l^o\ \varphi_0 \qquad y\ S\ H\ \Downarrow\ v\ \varphi_1 \qquad method(type(o), m) = m(z)\{s\} \\ S_0 = \{\mathsf{this} \mapsto (l^o\ \varphi_0), z \mapsto (v\ \varphi_1 \sqcup label(z))\} \qquad s\ \varphi \sqcup \varphi_0\ S_0\ H\ \Downarrow\ S_1\ H_1 \end{array}}{x.m(y)\ \varphi\ S\ H\ \Downarrow\ S\ H_1}$$

$$\text{[E-SEQ]}\ \frac{s_1\ \varphi\ E\ \Downarrow\ E_1 \qquad s_2\ \varphi\ E_1\ \Downarrow\ E_2}{s_1; s_2\ \varphi\ E\ \Downarrow\ E_2}$$

$$\text{[E-TRUE]}\ \frac{x\ E\ \Downarrow\ l\ \varphi_0 \qquad s_1\ \varphi \sqcup \varphi_0\ E\ \Downarrow\ E_1}{(\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)\ \varphi\ E\ \Downarrow\ E_1} \qquad \text{[E-FALSE]}\ \frac{x\ E\ \Downarrow\ \mathsf{null}\ \varphi_0 \qquad s_2\ \varphi \sqcup \varphi_0\ E\ \Downarrow\ E_2}{(\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)\ \varphi\ E\ \Downarrow\ E_2}$$

Figure 6 Big-step operational semantics.

flow by including the label of the source variable/field. All other rules are statement evaluations. [E-ASSIGN] and [E-STORE] update the stack and heap respectively (the notation $H[x{\mapsto}...]$ denotes l is updated in H) with an evaluated expression and its combined label including the label φ of implicit information flows passed to the statement. In both [E-LOAD] and [E-STORE], the security label of the receiver object is explicit included.

Object creation in [E-NEW] extends the stack and heap with newly allocated variable and object (labeled with its allocation site). The notation $H, l{\mapsto}...$ denotes extension of H. All fields are initially set to null with the label of the respective field. Note that the choice of fresh location allocated to new object is arbitrary in [E-NEW]. This allocation behavior is typically accommodated using standard renaming of locations over indistinguishable states. But for simplicity, like in [24], we also assume a deterministic allocator where fresh locations are allocated in order.

Method invocation in [E-CALL] dynamically looks up the target method to be called using the class name on the location. Since dynamic dispatches allow implicit information flow through the selection of methods based on the class type of the receiver object, the rule [E-CALL] propagates the combined label of the parent statement φ and the label of the location φ_0 to the evaluation of the body of the selected method. Similarly, the rules [E-TRUE] and [E-FALSE] allow implicit information flow through the conditional variable, where the label of the conditional variable is also propagated to the evaluation of the branches. The rule [E-SEQ] evaluates a composition of two statements in order: the latter is evaluated in the output state of the former.

4.2 Subject Reduction

The [CONCRETE STATE] rule defines a concrete state to be well-formed if the labels of values in the variables/fields respect the security police.

$$\frac{\begin{array}{c}\text{[CONCRETE STATE]}\\ \forall x \quad \cdot \quad S(x) = v\ \varphi \implies \varphi \trianglerighteq label(x) \quad label(x) \sqsubseteq \varphi \\ \forall l,f \quad \cdot \quad H(l)(f) = v\ \varphi \implies \varphi \trianglerighteq label(f) \quad label(f) \sqsubseteq \varphi\end{array}}{\vdash S\ H}$$

The correspondence between well-formed concrete state and abstract state is identified in [CORRESPONDENCE]. Informally, for every variable/field, the runtime value in the concrete state must be allocated within the corresponding points-to set in the abstract state, and its runtime label must also be subsumed by the corresponding static label. Thus, for every x in S, x must be defined in Γ such that $\Gamma(x)$ overapproximates $S(x)$. The condition on objects/fields is similar.

$$\frac{\begin{array}{c}\text{[CORRESPONDENCE]}\\ \vdash \Gamma\ \Sigma \qquad \vdash S\ H\\ \forall x \quad \cdot \quad S(x) = v\ \varphi_0 \implies \left\{\begin{array}{l}\Gamma(x) = \tau\ \varphi\\ \{v\} \subseteq \tau\\ \varphi_0 \sqsubseteq \varphi\end{array}\right.\\ \forall l^o,f \quad \cdot \quad H(l^o)(f) = v\ \varphi_0 \implies \left\{\begin{array}{l}\Sigma(o)(f) = \tau\ \varphi\\ \{v\} \subseteq \tau\\ \varphi_0 \sqsubseteq \varphi\end{array}\right.\end{array}}{\Gamma\ \Sigma \vdash S\ H}$$

Subset relation between points-to sets is extended to accommodate runtime values:

$$\frac{}{\{l^o\} \subseteq \{o\}}\ \text{[LOCATION]} \qquad \frac{}{\{\mathsf{null}\} \subseteq \tau}\ \text{[NULL]}$$

In order to prove noninterference by static analysis, we need the standard preservation theorem for subject reduction. We present only preservation for statements, because preservation for expressions is trivial—they only look up values from well-formed stack or heap.

▶ Theorem 1 (Preservation). For any static state A, dynamic states E_1, E_2, label φ and statement s:

$$\left.\begin{array}{c}A \vdash E_1\\ A \vdash s : \varphi\\ s\ \varphi\ E_1\ \Downarrow\ E_2\end{array}\right\} \implies A \vdash E_2$$

Proof. The proof is by structural induction on the derivation of $\Gamma\ \Sigma \vdash s : \varphi$. We present some interesting cases here.

Case [ASSIGN] Suppose the statement s is $x = e$. Consider the case when e is $y.f$. From [ASSIGN], for some τ, we have

$$\begin{array}{l}\Gamma(x) = \tau\ \varphi \text{ and}\\ \Gamma\ \Sigma \vdash y.f : \tau\ \varphi.\end{array} \tag{1}$$

Let $\Gamma(y) = \tau_0\ \varphi_0$. From [LOAD] and (1), for every $o \in \tau_0$ with $\Sigma(o)(f) = \tau_1\ \varphi_1$, we have:

$$\tau_1 \subseteq \tau, \tag{2}$$

$$\varphi_1 \sqsubseteq \varphi', \text{ where} \tag{3}$$

$$\varphi = \varphi' \sqcup \varphi_0. \tag{4}$$

Let $S_1(y) = l^o\ \beta$ and $H_1(l^o)(f) = v\ \alpha$. From the premise, we have:

$$\{o\} \subseteq \tau_0,$$

$$\{v\} \subseteq \tau_1, \text{ where } \Sigma(o)(f) = \tau_1\ \varphi_1, \tag{5}$$

$$\beta \sqsubseteq \varphi_0 \text{ and} \tag{6}$$

$$\alpha \sqsubseteq \varphi_1. \tag{7}$$

From $\vdash \Gamma\ \Sigma$ of the premise and from (1), we have:

$$\varphi \trianglerighteq \mathit{label}(x), \tag{8}$$

$$\mathit{label}(x) \sqsubseteq \varphi. \tag{9}$$

From the operational semantics we have $H_2 = H_1$ and S_2 is the same as S_1 except for x, i.e., $S_2(x) = v\ (\varphi \sqcup \alpha \sqcup \beta \sqcup \mathit{label}(x))$. To prove $\Gamma\ \Sigma \vdash S_2\ H_2$, we need to show:

$$\{v\} \subseteq \tau, \tag{10}$$

$$(\varphi \sqcup \alpha \sqcup \beta \sqcup \mathit{label}(x)) \sqsubseteq \varphi \text{ and} \tag{11}$$

$$\vdash S_2\ H_2. \tag{12}$$

From (5) and (2), we have (10). From (7), (3), (4) and from transitivity of $\sqsubseteq$ we have $\alpha \sqsubseteq \varphi$. From (6), (4) and from transitivity of $\sqsubseteq$ we have $\beta \sqsubseteq \varphi$. Then from (9), we have (11). From the premise we have $\mathit{label}(x) \sqsubseteq \varphi$. Hence we have (11). We now have $\varphi \sqcup \alpha \sqcup \beta \sqcup \mathit{label}(x) = \varphi$. Then from (8) and (9), we have (12). Thus $\Gamma\ \Sigma \vdash S_2\ H_2$.

Similar arguments hold good for the case when s is $x = y$.

Case [STORE] Suppose the statement s is $x.f = y$. From the premise and from [STORE], we have $\Gamma(x) = \tau_0\ \varphi_0$ such that for every $o \in \tau_0$, we have:

$$\Sigma(o)(f) = \tau_2\ \varphi_2, \tag{13}$$

$$\Gamma(y) = \tau_1\ \varphi_1,$$

$$\tau_1 \subseteq \tau_2, \tag{14}$$

$$\varphi \sqcup \varphi_0 \sqcup \varphi_1 \sqsubseteq \varphi_2. \tag{15}$$

Let $S_1(y) = v\ \alpha$ and $S_1(x) = l^o\ \beta$. From the premise we have:

$$\{v\} \subseteq \tau_1, \tag{16}$$

$$\alpha \sqsubseteq \varphi_1, \tag{17}$$

$$\{o\} \subseteq \tau_0,$$

$$\beta \sqsubseteq \varphi_0. \tag{18}$$

From the operational semantics we have $S_2 = S_1$ and H_2 is the same as H_1 except for the field f of l^o, i.e., $H_2(l^o)(f) = v\ (\varphi \sqcup \beta \sqcup \alpha \sqcup \mathit{label}(f))$, where . To prove $\Gamma\ \Sigma \vdash S_2\ H_2$, we need to show:

$$\{v\} \subseteq \tau_2, \tag{19}$$

$$(\varphi \sqcup \beta \sqcup \alpha \sqcup \mathit{label}(f)) \sqsubseteq \varphi_2 \text{ and} \tag{20}$$

$$\vdash S_2\ H_2. \tag{21}$$

From $\vdash \Gamma\ \Sigma$ of the premise and (13), we have

$$\varphi_2 \trianglerighteq \mathit{label}(f), \tag{22}$$

$$\mathit{label}(f) \sqsubseteq \varphi_2. \tag{23}$$

From (16) and (14) we have (19). From From (22) and (23), we have (21). Hence $\Gamma\ \Sigma \vdash S_2\ H_2$.

5 Intransitive Noninterference

In this section we define the security guarantees provided by our model, including the generalization of classical definitions of indistinguishability and noninterference. With a standard notion of noninterference [13], a program is modeled as a machine with inputs and outputs classified by security levels. The noninterference property for a confidentiality policy guarantees any sequence of lower-level inputs will produce the same lower-level outputs, regardless of what the higher-level inputs are. Intuitively, this ensures that an attacker (at lower-level) is not able to distinguish two computations from their outputs if they vary only in their secret (higher-level) inputs. Conversely, the noninterference property for an integrity policy guarantees any sequence of higher-level inputs will produce the same higher-level outputs, regardless of what the lower-level inputs are. This ensures that a system (at higher-level) is not able to distinguish two computations from their outputs if they vary only in their untrusted (lower-level) inputs.

Most of the existing literature assumes that the *base policy* for the information flow is transitive. Thus relaxing such a transitive policy requires ad-hoc downgrading actions, and there are various downgrading or declassification mechanisms as surveyed in [28]. The implementation of downgrading in object oriented programming languages is often application specific and cumbersome, such as the priviledged actions in the Java security model. The information flow policy of DAL is more general and flexible, which may be used to specify more precise requirement than what can be defined as a transitive noninterference policy. For instance, we may allow information to flow from a sensitive database to an API library, and public information to flow from the library to an application, but sensitive information is *not* allowed to flow from the database to the application through the library. This intransitive policy may be naturally specified in the DAL model without introducing any ad-hoc structures. On the other hand, using DAL as a framework for base policies is orthogonal to downgrading, i.e., one may add specific downgrading behaviours on top of DAL.

Before we present our noninterference property, we provide a number of auxiliary definitions used by the theorem. [INDISTINGUISHABILITY] identifies two states as indistinguishable for a specified label φ associated with a variable or a field. This can be intuitively thought of as consequences of a direct assignment. [STRONG INDISTINGUISHABILITY] identifies two states as indistinguishable if all variables or fields that may influence (hence the use of $\triangleright$) a given label must be the same in both states. The separation of the two rules are necessary to prove our noninterference theorem.

[INDISTINGUISHABILITY]

$$\frac{\begin{array}{rcl} \forall x & \cdot & label(x) = \varphi \implies S_1(x) = S_2(x) \\ \forall l, f & \cdot & label(f) = \varphi \implies H_1(l)(f) = H_2(l)(f) \end{array}}{S_1\ H_1 \overset{\varphi}{\approx} S_2\ H_2}$$

[STRONG INDISTINGUISHABILITY]

$$\frac{\begin{array}{rcl} \forall x & \cdot & label(x) \trianglerighteq \varphi \implies S_1(x) = S_2(x) \\ \forall l, f & \cdot & label(f) \trianglerighteq \varphi \implies H_1(l)(f) = H_2(l)(f) \end{array}}{S_1\ H_1 \overset{\triangleright\varphi}{\approx} S_2\ H_2}$$

▶ Theorem 2 (Noninterference). For any labels φ_0, φ, statement s, static state A and dynamic states E_1, E_2:

$$\left.\begin{array}{c} E_1 \overset{\rhd\varphi}{\approx} E_2 \\ A \vdash E_1 \\ A \vdash E_2 \\ A \vdash s : \varphi_0 \\ s\ \varphi_0\ E_1 \Downarrow E_3 \\ s\ \varphi_0\ E_2 \Downarrow E_4 \end{array}\right\} \implies E_3 \overset{\varphi}{\approx} E_4$$

Theorem 2 describes our intransitive noninterference theorem for a possibly intransitive security policy. Intuitively, Theorem 2 states that given a well-typed statement s, if executing s with a label φ_0 on two input states that are well-formed and indistinguishable on variables/fields having the same or a greater label than φ, it will result in output states that are indistinguishable under the same φ. This is because the variables/fields covered by φ can be influenced by only variables/fields having the same or a greater label but not by any other parts of the program. Note that the effect of executing the statement s in the context with the label φ_0 is captured in the states represented by E_3 and E_4, which need to be related by φ-indistinguishability.

Our notion of noninterference is generic and may accommodate other security policies. This is because the relation $\rhd$ in [STRONG INDISTINGUISHABILITY] can be possibly substituted with other information flow relation over different types of security labels.

Furthermore, this noninterference provides a unified guarantee of confidentiality and integrity. When we consider the label φ to be the system, integrity is given in the sense that the attacker (unauthorized source) cannot influence (send information to) the system. At the same time, we can also consider the label φ to represent the attacker, then confidentiality is given in the sense that the attacker (unauthorized sink) cannot be influenced by (receive information from) the system. Hence, φ in Theorem 2 captures both confidentiality and integrity requirements.

The proof of this theorem follows directly from Lemma 2, 3 and 4—Lemma 3 weakens strong indistinguishability of two input states; then Lemma 2 preserves indistinguishability based on dynamic labels from input states to output states over subject reduction; finally Lemma 4 weakens indistinguishability based on dynamic labels into indistinguishability on static labels of two output states.

Lemma 2 supports sequential composability by using a weakened version of indistinguishability provided by the definition [INDISTINGUISHABILITY BY DYNAMIC LABEL], which can be preserved over statement evaluation. It considers two states as indistinguishable if relevant variables/fields and all *values* that may influence variables in the given label must be the same in both states.

[INDISTINGUISHABILITY BY DYNAMIC LABEL]

$$\frac{\begin{array}{rl} \forall x\ . & \left.\begin{array}{c} S_1(x) = \_\ \varphi_1 \\ \varphi_1 \trianglerighteq \varphi \end{array}\right\} \implies S_1(x) = S_2(x) \\ \forall x\ . & \left.\begin{array}{c} S_2(x) = \_\ \varphi_2 \\ \varphi_2 \trianglerighteq \varphi \end{array}\right\} \implies S_1(x) = S_2(x) \\ \forall l,f\ . & \left.\begin{array}{c} H_1(l)(f) = \_\ \varphi_1 \\ \varphi_1 \trianglerighteq \varphi \end{array}\right\} \implies H_1(l)(f) = H_2(l)(f) \\ \forall l,f\ . & \left.\begin{array}{c} H_2(l)(f) = \_\ \varphi_2 \\ \varphi_2 \trianglerighteq \varphi \end{array}\right\} \implies H_1(l)(f) = H_2(l)(f) \end{array}}{S_1\ H_1 \overset{\rhd\varphi}{\sim} S_2\ H_2}$$

▶ Lemma 2 (Noninterference by Dynamic Label). For any labels φ_0, φ, statement s, static state A and dynamic states E_1, E_2:

$$\left.\begin{array}{c} E_1 \overset{\triangleright\varphi}{\sim} E_2 \\ A \vdash E_1 \\ A \vdash E_2 \\ A \vdash s : \varphi_0 \\ s\ \varphi_0\ E_1 \Downarrow E_3 \\ s\ \varphi_0\ E_2 \Downarrow E_4 \end{array}\right\} \Longrightarrow E_3 \overset{\triangleright\varphi}{\sim} E_4$$

Proof. The proof is by structural induction on the derivation of $s\ \varphi_0\ S_1\ H_1 \Downarrow S_3\ H_3$. We present some interesting cases here.

Case [E-STORE] Suppose the statement is $x.f = y$. From the operational semantics, we know that only the heap is changed. This means that $S_3 = S_1$ and $S_4 = S_2$. Let $S_1(x) = l_1\ \alpha_1$, $S_2(x) = l_2\ \alpha_2$, $S_1(y) = v_1\ \beta_1$ and $S_2(y) = v_2\ \beta_2$. From operational semantics, we have H_3 and H_4 to be the same as H_1 and H_2 respectively, except that $H_3(l_1)(f) = v_1\ \varphi_1$ and $H_4(l_2)(f) = v_2\ \varphi_2$ where $\varphi_1 = (\varphi_0 \sqcup \alpha_1 \sqcup \beta_1 \sqcup label(f))$ and $\varphi_2 = (\varphi_0 \sqcup \alpha_2 \sqcup \beta_2 \sqcup label(f))$. Suppose $\varphi_1 \sqcup \varphi_2 \trianglerighteq \varphi$. From Lemma 1, we have $\alpha_1 \trianglerighteq \varphi$, $\alpha_2 \trianglerighteq \varphi$, $\beta_1 \trianglerighteq \varphi$ and $\beta_2 \trianglerighteq \varphi$. Then from the premise we have $v_1 = v_2$, $\beta_1 = \beta_2$, $l_1 = l_2$ and $\alpha_1 = \alpha_2$. Hence $H_3 = H_4$. Thus $S_3\ H_3 \overset{\triangleright\varphi}{\sim} S_4\ H_4$.

Case [E-TRUE] Suppose the statement is $\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$. Let $S_1(x) = l_1\ \alpha_1$ and $S_2(x) = l_2\ \alpha_2$.

Suppose $\alpha_1 \sqcup \alpha_2 \trianglerighteq \varphi$. From the premise, we have $l_1 = l_2$ and $\alpha_1 = \alpha_2$. Consider the case when $l_1 \neq \mathsf{null}$ (same as $l_2 \neq \mathsf{null}$). The case for $l_1 = \mathsf{null}$ (same as $l_2 = \mathsf{null}$) follows on the same lines. Then from operational semantics we have:

$$s_1\ \varphi_0 \sqcup \alpha_1\ E_1 \Downarrow E_3 \text{ and} \tag{24}$$

$$s_1\ \varphi_0 \sqcup \alpha_2\ E_2 \Downarrow E_4. \tag{25}$$

Let $\Gamma(x) = \tau\ \gamma$. From the premise and the static semantics we have $A \vdash s_1 : \varphi_0 \sqcup \gamma$, where $\alpha_1 \sqsubseteq \gamma$ and $\alpha_2 \sqsubseteq \gamma$. Applying [SUB-STM], we have:

$$A \vdash s_1 : \varphi_0 \sqcup \alpha_1 \text{ and} \tag{26}$$

$$A \vdash s_1 : \varphi_0 \sqcup \alpha_2. \tag{27}$$

Now since $\alpha_1 = \alpha_2$, applying induction hypothesis on (24), (25) and (32) and using the premise, we have $E_3 \overset{\triangleright\varphi}{\sim} E_4$.

Suppose $\alpha_1 \sqcup \alpha_2 \not\trianglerighteq \varphi$. Then from Lemma 1 we have $\alpha_1 \not\trianglerighteq \varphi$ and $\alpha_2 \not\trianglerighteq$. This implies that:

$$\varphi_0 \sqcup \alpha_1 \not\trianglerighteq \varphi, \tag{28}$$

$$\varphi_0 \sqcup \alpha_2 \not\trianglerighteq \varphi. \tag{29}$$

Consider the case when $l_1 \neq \mathsf{null}$ and $l_2 = \mathsf{null}$. The other cases follow on the same lines. We have:

$$s_1\ \varphi_0 \sqcup \alpha_1\ E_1 \Downarrow E_3 \text{ and} \tag{30}$$

$$s_2\ \varphi_0 \sqcup \alpha_2\ E_2 \Downarrow E_4. \tag{31}$$

Let $\Gamma(x) = \tau\ \gamma$. From the premise and the static semantics we have $A \vdash s_1 : \varphi_0 \sqcup \gamma$ and $A \vdash s_2 : \varphi_0 \sqcup \gamma$, where $\alpha_1 \sqsubseteq \gamma$ and $\alpha_2 \sqsubseteq \gamma$. Applying [SUB-STM], we have:

$$A \vdash s_1 : \varphi_0 \sqcup \alpha_1 \text{ and} \tag{32}$$

$$A \vdash s_2 : \varphi_0 \sqcup \alpha_2. \tag{33}$$

Applying Lemma 5 on (32), (30), (28) and on (33), (31), (29) (and using the premise) we have:

$$E_1 \overset{\triangleright\varphi}{\sim} E_3, \tag{34}$$

$$E_2 \overset{\triangleright\varphi}{\sim} E_4. \tag{35}$$

From the premise, (34), (35), we finally have $E_3 \overset{\triangleright\varphi}{\sim} E_4$.

Case [E-CALL] Suppose the statement s is $x.m(y)$. From the operational semantics, we know that $S_3 = S_1$ and $S_4 = S_2$. Let $S_1(x) = l_1^{o_1}\ \alpha_1$, $S_2(x) = l_2^{o_2}\ \alpha_2$, $S_1(y) = v_1\ \beta_1$, $S_2(y) = v_2\ \beta_2$, $\Gamma(x) = \tau_x\ \varphi_x$ and $\Gamma(y) = \tau_y\ \varphi_y$. From (100) and (101), we have $\{o_1, o_2\} \subseteq \tau_x$ and $\alpha_1 \sqsubseteq \varphi_x$, $\alpha_2 \sqsubseteq \varphi_x$, $\beta_1 \sqsubseteq \varphi_y$ and $\beta_2 \sqsubseteq \varphi_y$.
Let $S_1^0 = \{\mathsf{this} \mapsto (l_1^{o_1}\ \alpha_1), z \mapsto (v_1\ \beta_1 \sqcup \mathit{label}(z))\}$. Let $S_2^0 = \{\mathsf{this} \mapsto (l_2^{o_2}\ \alpha_2), z \mapsto (v_2\ \beta_2 \sqcup \mathit{label}(z))\}$. Let

$$\mathit{method}(\mathit{type}(o_1), m) = m(z)\{s_1\},$$

$$\mathit{method}(\mathit{type}(o_2), m) = m(z)\{s_2\}.$$

From operational semantics, we have:

$$s_1\ (\varphi_0 \sqcup \alpha_1)\ S_1^0\ H_1\ \Downarrow\ S_3^0\ H_3 \text{ and} \tag{36}$$

$$s_2\ (\varphi_0 \sqcup \alpha_2)\ S_2^0\ H_2\ \Downarrow\ S_4^0\ H_4. \tag{37}$$

Let $\Gamma_0 = \{\mathsf{this} \mapsto (\tau_x\ \varphi_x), z \mapsto (\tau_y\ \varphi_y)\}$. From [CALL] we have $\Gamma_0\ \Sigma \vdash s_i : \varphi_0 \sqcup \varphi_x$, $1 \le i \le 2$ and [SUB-STM] of static semantics, we have:

$$\Gamma_0\ \Sigma \vdash s_1 : \varphi_0 \sqcup \alpha_1, \tag{38}$$

$$\Gamma_0\ \Sigma \vdash s_2 : \varphi_0 \sqcup \alpha_2. \tag{39}$$

From the premise, we have:

$$\Gamma_0\ \Sigma \vdash S_1^0\ H_1, \tag{40}$$

$$\Gamma_0\ \Sigma \vdash S_2^0\ H_2, \tag{41}$$

$$S_1^0\ H_1\ \overset{\triangleright\varphi}{\sim}\ S_2^0\ H_2. \tag{42}$$

Suppose $\alpha_1 \sqcup \alpha_2 \trianglerighteq \varphi$. From the premise, we have that $l_1^{o_1} = l_2^{o_2}$, $\alpha_1 = \alpha_2$. This means that the same method is invoked and $s_1 = s_2$. Then applying induction hypothesis on (42), (40), (41), (38), (36), (37), we have $S_3^0\ H_3\ \overset{\triangleright\varphi}{\sim}\ S_4^0\ H_4$. Because the stack is not changed by the statement $x.m(y)$, we have $S_3\ H_3\ \overset{\triangleright\varphi}{\sim}\ S_4\ H_4$.

Suppose $\alpha_1 \sqcup \alpha_2 \not\trianglerighteq \varphi$. Then we have $\alpha_1 \not\trianglerighteq \varphi$ and $\alpha_2 \not\trianglerighteq \varphi$. From Lemma 1 we have:

$$\varphi_0 \sqcup \alpha_1 \not\trianglerighteq \varphi, \tag{43}$$

$$\varphi_0 \sqcup \alpha_2 \not\trianglerighteq \varphi. \tag{44}$$

Then applying Lemma 5 on (38), (40), (36), (43) and (39), (41), (37), (44) we have:

$$S_1^0\ H_1\ \overset{\triangleright\varphi}{\sim}\ S_3^0\ H_3, \tag{45}$$

$$S_2^0\ H_2\ \overset{\triangleright\varphi}{\sim}\ S_4^0\ H_4. \tag{46}$$

Then from (42), (45), (46), we have $S_3^0\ H_3\ \overset{\triangleright\varphi}{\sim}\ S_4^0\ H_4$. Because the stack is not changed by the statement $x.m(y)$, we have $S_3\ H_3\ \overset{\triangleright\varphi}{\sim}\ S_4\ H_4$. Hence proved.

From the dynamic semantics, the labels on values relate to the reading of values ([E-VAR] and [E-LOAD]) while the labels on variables/fields relate to both writing and reading into a variable/field ([E-ASSIGN] and [E-STORE] in addition to [E-VAR] and [E-LOAD]). [STRONG INDISTINGUISHABILITY] constrains the labels on *variables* that always cover the values that may be read from them, therefore implying the indistinguishability in [INDISTINGUISHABILITY BY DYNAMIC LABEL].

▶ Lemma 3 (Weakening Strong Indistinguishability). For any dynamic states E_1, E_2 and label φ:

$$\left.\begin{array}{l} E_1\ \overset{\triangleright\varphi}{\approx}\ E_2 \\ \vdash E_1 \\ \vdash E_2 \end{array}\right\} \implies E_1\ \overset{\triangleright\varphi}{\sim}\ E_2$$

▶ Lemma 4 (Weakening Indistinguishability by Dynamic Label). For any dynamic states E_1, E_2 and label φ:

$$\left.\begin{array}{c} E_1 \overset{\triangleright\varphi}{\sim} E_2 \\ \vdash E_1 \\ \vdash E_2 \end{array}\right\} \Longrightarrow E_1 \overset{\varphi}{\approx} E_2$$

Lemma 5 characterizes the part of the concrete state that is indistinguishable.

▶ Lemma 5 (Side Effect). For any statement s, labels φ_s, φ, static state A and dynamic states E_1, E_2:

$$\left.\begin{array}{c} A \vdash s : \varphi_s \\ A \vdash E_1 \\ s\ \varphi_s\ E_1 \Downarrow E_2 \\ \varphi_s \not\sqsubseteq \varphi \end{array}\right\} \Longrightarrow E_1 \overset{\triangleright\varphi}{\sim} E_2$$

Proof. The proof is by structural induction on the derivation of $s\ \varphi_s\ S_1\ H_1 \Downarrow S_2\ H_2$. We present some interesting cases here.

Case [E-NEW] Suppose the statement s is $x = \mathsf{new}^o c$. From the operational semantics, we have S_1 the same as S_2, except for a new variable $x \notin dom(S_1)$, i.e., $S_2(x) = l^o\ (\varphi_s \sqcup label(x))$ for some $l^o \notin dom(H_1)$, and $H_2 = H_1, l^o \mapsto (\overline{f \mapsto (\mathsf{null}\ label(f))})$. from the definition of [INDISTINGUISHABILITY BY DYNAMIC LABEL] and since $x \notin dom(S_1)$ and $l^o \notin dom(H_1)$, we vacuously have $S_1\ H_1 \overset{\triangleright\varphi}{\sim} S_2\ H_2$.

Case [E-SEQ] Suppose the statement s is $s_1; s_2$. From the operational semantics, we have: $s_1\ \varphi_s\ E_1 \Downarrow E_1'$ and $s_2\ \varphi_s\ E_1' \Downarrow E_2$. From the premise, we have $A \vdash s_1 : \varphi_s$ and $A \vdash s_2 : \varphi_s$. Applying induction hypothesis, we have $E_1 \overset{\triangleright\varphi}{\sim} E_1'$ and $E_1' \overset{\triangleright\varphi}{\sim} E_2$. Hence $E_1 \overset{\triangleright\varphi}{\sim} E_2$.

6 Related Work

Information flow security has been an active research topic since the 1970s [8, 9]. The general notion of information flow based on noninterference is introduced in [13]. Information flow has been considered in the context of programming languages since the 1990s by Volpano et al. [32]. A survey for early development on language-based information flow security can be found in [27]. A classification of extensions such as secure downgrading and declassification is discussed in [28].

In this section we review the literature related to our work in two categories: security models and extensions to Java/CLR's access control mechanism, and other language-based security models with various notions for downgrading or intransitive information flow. Various notions of intransitive noninterference have been used in literature to refer to extended transitive noninterference theorems that accommodate various downgrading policies [25]. Downgrading techniques are orthogonal to the work presented in this paper. Although our security model does not use any downgrading policies, it is possible to adopt such polices to support runtime updates on security labels and information flow relations.

6.1 Extensions to Java/CLR's Access Control Model

The stack inspection model has been formalized in [12] which does not provide a security guarantee. History-based access control model [1] is proposed to prevent undesired influence from authorized code to security-sensitive code. When a security-sensitive code is accessed, all the code previously executed (and not just the code currently on the stack) must be sufficiently authorized to access that resource, regardless of the fact that some of that code

may not be responsible for the resource access— potentially rejecting many good programs. Moreover, it does not prevent undesired release of sensitive information to unauthorized code that has not be executed (thus not tracked in the history). Our type system tracks the history of information flows instead of calls, and guards against both undesired influence and release.

Information-based access control (IBAC) [24] presents an access control policy that is stronger than stack inspection but does not demand the stringent requirements of the history-based model. However, IBAC considers only integrity of specified tainted sources. They also do not present any static analysis that provides guarantees for the policy. The separation of accessibility and capability is fundamental to our security model in order to enforce both integrity and confidentiality requirements.

A type system to address the issue of integrity and confidentiality in the context of stack inspection has been developed [3]. The type system preserves both confidentiality and integrity in a Java-like language with defined access rules and permission checks. Noninterference is proved with respect to confinement and indistinguishability in a transitive H-L policy. In effect, they use permission checks to allow safe downgrading in the classical information flow security models where the security level of data is given. In contrast, our security lattice can provide different confidentiality and integrity guarantees with DAL and thus is more closely aligned with Java's access control model.

6.2 Other Language-based Security Models

Jif [6, 21], which follows a decentralized label model [22], is one of the first language implementations to enforce information flow security in object-oriented programming languages. At high-level, Jif provides a transitive information flow framework via multiple ownership of labeled information. The exceptions to this transitive policy are introduced via programmatic constructs of declassification and downgrading. The essential contrasting feature of our security model is the expressive power of its policies. By relaxing the restriction of transitivity, we can handle the *where* dimension [28] in downgrading without special programming constructs. Note that each Jif label denotes a set of policies. Since our goal is to support Java/CLR's access control model where permissions are specified on code blocks, our policy can be considered as a restriction of Jif where labels specify a single policy.

CapPCF [10] from Dimoulas et al. is an extension of Plotkin's PCF with capabilities as declarative policies for access control and integrity. They apply higher-order contracts for dynamic enforcement and a security type system for static guarantee. However, they do not discuss intransitive policies and also do not have a uniform treatment of integrity and confidentiality. Drossopoulou and Noble [11] develop a specification language to formally express security properties using capabilities separate to the program implementations. This language could be used to address confidentiality and integrity but it relies on the specifier to encode these requirements as part of the specification.

Disjunction Category (DC) labels [31] is a label format that classifies data sensitivities in information flow control systems, with privileges that enable downgrading specified as declassification (for confidentiality policies) and endorsement (for integrity policies). [33] extends the DC-label system with bounded privileges and robust privileges. Boundedness specifies predefined upper-bound and lower-bound for downgrading, and robustness disallows downgrading if data integrity is violated. Though the notion of downgrading (of confidential information) is not explicitly specified in our policy, the effects of boundedness and robustness can be encoded in our dual labeled model. For instance, the passage of confidential information has to restricted by the integrity requirements. Like our approach, their DC-labels combine

both confidentiality and integrity, in the way of $\langle C_1, I_1\rangle \sqsubseteq \langle C_2, I_2\rangle$ if $C_1 \sqsubseteq^C C_2$ and $I_1 \sqsubseteq^I I_2$. However, their policy relation (can-flow-to relation) is transitive and forms a standard security lattice.

A type system to specify robust declassification [23] considers when a trusted entity controls what can be downgraded. Our approach defines a not necessarily transitive information flow system where access control permissions are assigned to code fragments. Following the classification developed by Sabelfeld and Sands [28], our approach uses the "where" specification while the robust declassification approach uses the "who" specification.

Austin and Flanagan [2] have a permissive dynamic information that is a weakened noninterference policy under the H-L two security level setting. In that policy, an assignment of an L value under the guarded of H needs not be rejected immediately—the rejection is postponed until that L value is actually used. This policy has been extended in [4] to a general lattice. On one hand, like our approach, they guarantee termination-insensitive noninterference with a proof. However, since our approach is based on DAL, the policy we apply is not necessarily transitive, thus it is more general than a lattice-based information flow policy.

Paragon [5] is a language to express a variety of information policies, although the focus is on declassification, for a given Java program. It is expressed as modifiers for methods/fields and as predicates (also called locks). Locks can be opened (i.e., set to true) or closed (i.e., set to false). One can also expect that a lock should be open whenever a method is called. Such policies are enforced in phases, including static type checking and policy constraint solving.

Rushby [26] considers intransitive policies and develops a noninterference theorem in that context. The system Rushby worked on is state based, which accepts user inputs and produces observable outputs. Therefore, the notion of noninterference is input-output based, and moreover, the intransitive feature in [26] is expressed as channel control, which has been classified as a type of (where-based) downgrading [28]. For instance, if $H \rhd D$ and $D \rhd L$ but $H \not\rhd L$, then H's behaviour is allowed to interfere with L only if D is involved (i.e., D actively participates as a downgrader). If adapted in our setting, sensitive information is allowed to be released only via certain intermediate classes/methods, which is intuitively weaker than what our current model can enforce.

Information flow analysis for Java has also been considered in [16, 15, 19]. They focus on the analysis techniques for scalability and precision, and generally do not take the Java security model and permissions into account. In contrast, we focus on security models for programming languages. Our type system is designed to demonstrate how to enforce the security model statically. Although it uses a context-insensitive and flow-insensitive points-to analysis, it is possible to exploit various types of sensitivity to achieve desired scalability and precision.

Ownership types [7] and its variants support instance-level information hiding by providing a statically enforceable object encapsulation model based on an ownership relation. They restrict object accessibility to protect encapsulated objects from being exposed to external contexts, yielding a fine-grained object-based access control model [20]. Although they provide a sense of confidentiality where runtime objects are considered as security levels that are ordered by their ownership relation, these type systems do not consider information flow security in general.

7 Conclusion

The stack-based access control model in Java and CLR is often used with the intent of preventing unwanted information flow between untrusted code and security-sensitive code. However, access control is inherently weak such that violations caused by unauthorized information flow are not necessarily detected. We have presented a security model based on DAL to directly capture the security intent on information flow, which tracks information flow between code by reusing access control specification on them. Significantly, our information flow security model retains the required intransitivity in applications that rely on the access control model. We develop a generic type system to enforce the security model and provide a new notion of intransitive noninterference theorem. The intransitive noninterference generalizes the standard notion of transitive noninterference, and may support a wider range of security labels which may not necessarily form a lattice. This provides both confidentiality and integrity guarantees while allowing cyclic information flows among code with different security labels. The proof of the noninterference theorem relies on a generalization of the usual definition of indistinguishability, which considers both values and variables/fields to account for reading from and writing to the variables/fields.

References

1. Martin Abadi and Cedric Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, 2003.
2. Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12. ACM, 2010.
3. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, March 2005.
4. Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 15:15–15:24. ACM, 2014.
5. Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *Asian Symposium on Programming Languages and Systems (APLAS)*, number 8301 in LNCS, pages 217–232. Springer, 2013.
6. Stephen Chong, Andrew C. Myers, K. Vikram, and Lantian Zheng. Jif reference manual. `http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`, 2009.
7. David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64. ACM, 1998.
8. Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
9. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
10. Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium*, pages 3–17. IEEE, 2014.
11. Sophia Drossopoulou and James Noble. How to break the bank: Semantics of capability policies. In *Integrated Formal Methods (IFM)*, volume 8739 of *LNCS*, pages 18–35. Springer, 2014.

12 Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 307–318, 2002.

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

14 Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *1st USENIX Symposium on Internet Technologies and Systems, USITS'97, Monterey, California, USA, December 8-11, 1997*, 1997.

15 Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs - A practical guide. In *Software Engineering (Workshops)*, volume 215 of *LNI*, pages 123–138. GI, 2013.

16 Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

17 Norman Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–38, 1988.

18 Secure coding guidelines for Java SE. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`, 2015. Document version 5.1.

19 Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. Flowtwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 98–108, 2014.

20 Yi Lu and John Potter. On ownership and accessibility. In *the 20th European Conference on Object-Oriented Programming*, pages 99–123, 2006.

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

22 Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

23 Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

24 Marco Pistoia, Anindya Banerjee, and David A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *IEEE Symposium on Security and Privacy*, pages 149–163. IEEE Computer Society, 2007.

25 A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *CSFW*, pages 228–238, 1999.

26 John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI International: Computer Science Laboratory, 1992.

27 A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1)*, 2003.

28 Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2007.

29 Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, November 1993.

30 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, April 2015.

31 Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems*, pages 223–239, 2011.

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

33 Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. It's my privilege: Controlling downgrading in DC-labels. In *Security and Trust Management (STM)*, volume 9331 of *LNCS*, pages 203–219. Springer, 2015.

A Proofs for Properties in Section 4

We give the proof of Theorm 1. From the premise, we have:

$$\Gamma\ \Sigma \vdash s : \varphi \tag{47}$$

$$s\ \varphi\ S\ H\ \Downarrow\ S_1\ H_1 \tag{48}$$

$$\Gamma\ \Sigma \vdash S\ H \tag{49}$$

Proof. The proof is by structural induction on the derivation of $\Gamma\ \Sigma \vdash s : \varphi$. The rules [ASSIGN], [STORE] and [NEW] form the base cases. Rules [CALL], [IF] and [SEQ] form the inductive cases.

Case [ASSIGN] Suppose the statement s is $x = e$. Consider the case when e is $y.f$. From [ASSIGN], for some τ, we have

$$\Gamma(x) = \tau\ \varphi \text{ and} \tag{50}$$

$$\Gamma\ \Sigma \vdash y.f : \tau\ \varphi.$$

Let $\Gamma(y) = \tau_0\ \varphi_0$. From [LOAD] and (50), for every $o \in \tau_0$ with $\Sigma(o)(f) = \tau_1\ \varphi_1$, we have:

$$\tau_1 \subseteq \tau, \tag{51}$$

$$\varphi_1 \sqsubseteq \varphi', \text{ where} \tag{52}$$

$$\varphi = \varphi' \sqcup \varphi_0. \tag{53}$$

Let $S(y) = l^o\ \beta$ and $H(l^o)(f) = v\ \alpha$. From (49), we have:

$$\{o\} \subseteq \tau_0,$$

$$\{v\} \subseteq \tau_1, \text{ where } \Sigma(o)(f) = \tau_1\ \varphi_1, \tag{54}$$

$$\beta \sqsubseteq \varphi_0 \text{ and} \tag{55}$$

$$\alpha \sqsubseteq \varphi_1. \tag{56}$$

From $\vdash \Gamma\ \Sigma$ of (49) and from (50), we have:

$$\varphi \trianglerighteq \mathit{label}(x), \tag{57}$$

$$\mathit{label}(x) \sqsubseteq \varphi. \tag{58}$$

From the operational semantics we have $H_1 = H$ and S_1 is the same as S except for x, i.e., $S_1(x) = v\ (\varphi \sqcup \alpha \sqcup \beta \sqcup \mathit{label}(x))$. To prove $\Gamma\ \Sigma \vdash S_1\ H_1$, we need to show:

$$\{v\} \subseteq \tau, \tag{59}$$

$$(\varphi \sqcup \alpha \sqcup \beta \sqcup \mathit{label}(x)) \sqsubseteq \varphi \text{ and} \tag{60}$$

$$\vdash S_1\ H_1. \tag{61}$$

From (54) and (51), we have (59). From (56), (52), (53) and from transitivity of $\sqsubseteq$ we have $\alpha \sqsubseteq \varphi$. From (55), (53) and from transitivity of $\sqsubseteq$ we have $\beta \sqsubseteq \varphi$. Then from (58) we have (60). We now have $\varphi \sqcup \alpha \sqcup \beta \sqcup \mathit{label}(x) = \varphi$. Then from (57) and (58), we have (61). Thus $\Gamma\ \Sigma \vdash S_1\ H_1$.

Similar arguments hold good for the case when s is $x = y$.

Case [STORE] Suppose the statement s is $x.f = y$. From [STORE] and (47), we have $\Gamma(x) = \tau_0\ \varphi_0$ such that for every $o \in \tau_0$, we have:

$$\Sigma(o)(f) = \tau_2\ \varphi_2, \tag{62}$$

$$\Gamma(y) = \tau_1\ \varphi_1,$$

$$\tau_1 \subseteq \tau_2, \tag{63}$$

$$\varphi \sqcup \varphi_0 \sqcup \varphi_1 \sqsubseteq \varphi_2, \tag{64}$$

Let $S(y) = v\ \alpha$ and $S(x) = l^o\ \beta$. From the (49) we have:

$$\{v\} \subseteq \tau_1, \tag{65}$$

$$\alpha \sqsubseteq \varphi_1, \tag{66}$$

$$\{o\} \subseteq \tau_0,$$

$$\beta \sqsubseteq \varphi_0. \tag{67}$$

From the operational semantics we have $S_1 = S$ and H_1 is the same as H except for the field f of l^o, i.e., $H_1(l^o)(f) = v\ (\varphi \sqcup \beta \sqcup \alpha \sqcup label(f))$, where . To prove $\Gamma\ \Sigma \vdash S_1\ H_1$, we need to show:

$$\{v\} \subseteq \tau_2, \tag{68}$$

$$(\varphi \sqcup \beta \sqcup \alpha \sqcup label(f)) \sqsubseteq \varphi_2 \text{ and} \tag{69}$$

$$\vdash S_1\ H_1. \tag{70}$$

From $\vdash \Gamma\ \Sigma$ of (49) and (62), we have

$$\varphi_2 \trianglerighteq label(f), \tag{71}$$

$$label(f) \sqsubseteq \varphi_2. \tag{72}$$

From (65) and (63) we have (68). From (64), (67), (66), (72) and from transitivity of $\sqsubseteq$ we have (69). From (71) and (72), we have (70). Hence $\Gamma\ \Sigma \vdash S_1\ H_1$.

Case [NEW] Suppose the statement s is $x = \mathsf{new}^o c$. From the operational semantics we have $x \notin dom(S)$, $l^o \notin dom(H)$, $S_1(x) = l^o\ \varphi$ and $H_1(l^o) = \overline{(f \mapsto (\mathsf{null}\ label(f)))}$. From the rule [NEW] and (47), we have $\Gamma(x) = \tau\ \varphi$ with $o \in \tau$. We have $\{l^o\} \subseteq \{o\}$. For a field f, let $\Sigma(o)(f) = \tau'\ \alpha$. Then from $\vdash \Gamma\ \Sigma$ of (49), we have $\varphi \trianglerighteq label(x)$, $label(x) \sqsubseteq \varphi$, $\alpha \trianglerighteq label(f)$ and $label(f) \sqsubseteq \alpha$. We also have $\{l^o\} \subseteq \{o\} \subseteq \tau$ and $\mathsf{null} \subseteq \tau'$. Thus $\Gamma\ \Sigma \vdash S_1\ H_1$.

Case [CALL] Suppose the statement s is $x.m(y)$. From (47) and [CALL], we have:

$$\Gamma\ \Sigma \vdash x : \tau_0\ \varphi_0 \text{ and}$$

$$\Gamma\ \Sigma \vdash y : \tau_1\ \varphi_1$$

such that for every $o \in \tau_0$,

$$\Gamma_0\ \Sigma \vdash s_1 : \varphi \sqcup \varphi_0, \tag{73}$$

where $\Gamma_0 = \{\mathsf{this} \mapsto (\tau_0\ \varphi_0), z \mapsto (\tau_1\ \varphi_1)\}$, and
$method(type(o), m) = m(z)\{s_1\}$. From (49) we have

$$\Gamma_0\ \Sigma \vdash S_0\ H \tag{74}$$

where $S_0 = \{\mathsf{this} \mapsto (l^o\ \alpha_0), z \mapsto (v\ \alpha_1)\}$ such that $S(x) = l^o\ \alpha_0$, $S(y) = v\ \alpha_1$, $\alpha_0 \sqsubseteq \varphi_0$, $\alpha_1 \sqsubseteq \varphi_1$ and

$$s_1\ \varphi \sqcup \alpha_0\ S_0\ H\ \Downarrow\ S_1\ H_1. \tag{75}$$

From [SUBST], we have

$$\Gamma_0\ \Sigma \vdash s_1 : \varphi \sqcup \alpha_0. \tag{76}$$

Applying induction hypothesis on (74), (75), (76), we have

$$\Gamma_0\ \Sigma \vdash S_1\ H_1. \tag{77}$$

This means that $(\Gamma_0 \cup \Gamma)\ \Sigma \vdash (S_1 \cup S)\ H_1$. Hence $\Gamma\ \Sigma \vdash S\ H_1$.

Case [IF] Suppose the statement s is $\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$. From (47) and [IF] we have:

$$\Gamma\ \Sigma \vdash s_1 : \varphi \sqcup \varphi_0, \tag{78}$$

$$\Gamma\ \Sigma \vdash s_2 : \varphi \sqcup \varphi_0 \text{ and} \tag{79}$$

$$\Gamma\ \Sigma \vdash x : \tau_0\ \varphi_0$$

There are two sub-cases: $S(x) = l\ \alpha_0$ and $S(x) = \mathsf{null}\ \alpha_1$. We only deal with the sub-case $S(x) = l\ \alpha_0$. The other sub-case runs on similar lines. From (49) we have $\{l\} \subseteq \tau_0$ and $\alpha_0 \sqsubseteq \varphi_0$. Then from [SUBST] and (78) we have

$$\Gamma\ \Sigma \vdash s_1 : \varphi \sqcup \alpha_0. \tag{80}$$

From the operational semantics, we have

$$s_1\ (\varphi \sqcup \alpha_0)\ S\ H\ \Downarrow\ S_1\ H_1. \tag{81}$$

Applying induction hypothesis on (49), (80), (81), we have $\Gamma\ \Sigma \vdash S_1\ H_1$.

Case [SEQ] Suppose the statement s is $s_1; s_2$. Then from (47) and [SEQ] we have $\Gamma\ \Sigma \vdash s_1 : \varphi$ and $\Gamma\ \Sigma \vdash s_2 : \varphi$. From the operational semantics we have $s_1\ \varphi\ S\ H\ \Downarrow\ S_2\ H_2$ and $s_2\ \varphi\ S_2\ H_2\ \Downarrow\ S_1\ H_1$. Then from (49) and induction hypothesis we have $\Gamma\ \Sigma \vdash S_2\ H_2$ and $\Gamma\ \Sigma \vdash S_1\ H_1$. Hence proved.

The preservation theorem ensures that output state from a well-typed statement with a well-formed input state is also well-formed, which is necessary for the noninterference theorem given in the next section.

B Proofs for Properties in Section 5

B.1 Proof of Lemma 5

From the premise, we have:

$$\Gamma\ \Sigma \vdash s : \varphi_s \tag{82}$$

$$\Gamma\ \Sigma \vdash S_1\ H_1 \tag{83}$$

$$s\ \varphi_s\ S_1\ H_1\ \Downarrow\ S_2\ H_2 \tag{84}$$

$$\varphi_s \not\sqsubseteq \varphi \tag{85}$$

The proof is by structural induction on the derivation of $s\ \varphi_s\ S_1\ H_1\ \Downarrow\ S_2\ H_2$. The rules [E-ASSIGN], [E-STORE] and [E-NEW] form the base cases. Rules [E-CALL], [E-TRUE], [E-FALSE] and [E-SEQ] form the inductive cases.

Case [E-ASSIGN] Suppose the statement s is $x = e$. Consider the case when e is $y.f$. Let $S_1(y) = l\ \alpha$ and $H_1(l)(f) = v\ \beta$. From operational semantics we have $H_1 = H_2$ and S_1 is the same as S_2 except for x, i.e., $S_2(x) = v_2\ \varphi_2$ where $\varphi_2 = \varphi_s \sqcup \alpha \sqcup \beta \sqcup label(x)$. From Lemma 1 and (85) we have

$$\varphi_2 \not\sqsubseteq \varphi. \tag{86}$$

Let $S_1(x) = v_1\ \varphi_1$. Again from Lemma 1 and (86) we have $\varphi_1 \sqcup \varphi_2 \not\trianglerighteq \varphi$. Then by the definition of [INDISTINGUISHABILITY BY DYNAMIC LABEL], there is no obligation to be met by $S_1(x)$ and $S_2(x)$. Hence $S_1\ H_1 \overset{\triangleright\varphi}{\sim} S_2\ H_2$.

The case for $x = y$ runs on similar lines.

Case [E-STORE] Suppose the statement s is $x.f = y$. Let $S_1(x) = l\ \alpha$ and $S(y) = v\ \beta$. From operational semantics we have $S_1 = S_2$ and H_1 is the same as H_2 except for the value of field f at location l, i.e., $H_2(l)(f) = v_2\ \varphi_2$, where $\varphi_2 = \varphi_s \sqcup \alpha \sqcup \beta \sqcup label(f)$. From Lemma 1 and (85) we have

$$\varphi_2 \not\trianglerighteq \varphi. \tag{87}$$

Let $H_1(l)(f) = v_1\ \varphi_1$. Again from Lemma 1 and (87) we have $\varphi_1 \sqcup \varphi_2 \not\trianglerighteq \varphi$. Then by the definition of [INDISTINGUISHABILITY BY DYNAMIC LABEL], there is no obligation to be met by $H_1(l)(f)$ and $H_2(l)(f)$. Hence $S_1\ H_1 \overset{\triangleright\varphi}{\sim} S_2\ H_2$.

Case [E-NEW] Suppose the statement s is $x = \mathsf{new}^o c$. From the operational semantics, we have S_1 the same as S_2, except for a new variable $x \notin dom(S_1)$, i.e., $S_2(x) = l^o\ (\varphi_s \sqcup label(x))$ for some $l^o \notin dom(H_1)$, and $H_2 = H_1, l^o \mapsto (\overline{f \mapsto (\mathsf{null}\ label(f))})$. From the definition of [INDISTINGUISHABILITY BY DYNAMIC LABEL] and since $x \notin dom(S_1)$ and $l^o \notin dom(H_1)$, we vacuously have $S_1\ H_1 \overset{\triangleright\varphi}{\sim} S_2\ H_2$.

Case [E-SEQ] Suppose the statement s is $s_1; s_2$. From the operational semantics, we have:

$$s_1\ \varphi_s\ E_1 \Downarrow E_1' \text{ and} \tag{88}$$

$$s_2\ \varphi_s\ E_1' \Downarrow E_2. \tag{89}$$

Applying static semantics on (82), we have:

$$A \vdash s_1 : \varphi_s \text{ and} \tag{90}$$

$$A \vdash s_2 : \varphi_s. \tag{91}$$

Then applying induction hypothesis on (90), (83), (88), (85) and (91), (83), (89), (85), we have:

$E_1 \overset{\triangleright\varphi}{\sim} E_1'$,

$E_1' \overset{\triangleright\varphi}{\sim} E_2$.

Hence $E_1 \overset{\triangleright\varphi}{\sim} E_2$.

Case [E-TRUE] Suppose the statement s is $\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$. Let $S_1(x) = l\ \alpha$ where $l \neq \mathsf{null}$. The arguments when $l = \mathsf{null}$ are on the same lines. From the operational semantics we have:

$$s_1\ \varphi_s \sqcup \alpha\ E_1 \Downarrow E_2. \tag{92}$$

Let $\Gamma(x) = \tau_0\ \varphi_0$. From $\vdash \Gamma\ \Sigma$ of (83), we have $\alpha \sqsubseteq \varphi_0$. From static semantics of [IF] and [SUB-STM], we have:

$$A \vdash s_1 : \varphi_s \sqcup \alpha. \tag{93}$$

From Lemma 1, we have:

$$\varphi_s \sqcup \alpha \not\trianglerighteq \varphi. \tag{94}$$

Now by applying induction hypothesis on (93), (83), (92), (94), we have $E_1 \overset{\triangleright\varphi}{\sim} E_2$.

Case [E-FALSE] Similar to the case of [E-TRUE].

Case [E-CALL] Suppose the statement s is $x.m(y)$. Let $S_1(x) = l^o\ \alpha$, $S(y) = v\ \beta$ and $method(type(o), m) = m(z)\{s_1\}$. From the operational semantics we have:

$$s_1\ \varphi_s \sqcup \alpha\ S_0\ H_1 \Downarrow S_0'\ H_2 \tag{95}$$

where $S_0 = \{\mathsf{this} \mapsto l^o\ \alpha, z \mapsto v\ \beta\}$. Let $\Gamma(x) = \tau_0\ \varphi_0$ and $\Gamma(y) = \tau_1\ \varphi_1$. We have $\{l^o\} \subseteq \tau_0$, $\alpha \sqsubseteq \varphi_0$, $\{v\} \subseteq \tau_1$ and $\beta \sqsubseteq \varphi_1$. Let $\Gamma_0 = \{\mathsf{this} \mapsto \tau_0\ \varphi_0, z \mapsto \tau_1\ \varphi_1\}$. Then we have:

$$\Gamma_0\ \Sigma \vdash S_0\ H_1. \tag{96}$$

From [CALL] and [SUB-STM] of static semantics, we have:

$$\Gamma_0\ \Sigma \vdash s_1 : \varphi_s \sqcup \alpha. \tag{97}$$

Applying Lemma 1 on (85), we have:

$$\varphi_s \sqcup \alpha \not\trianglerighteq \varphi. \tag{98}$$

Now by applying induction hypothesis on (97), (96), (95) and (98), we have $S_0\ H_1 \overset{\triangleright\varphi}{\sim} S_0'\ H_2$. Hence we have $S_1\ H_1 \overset{\triangleright\varphi}{\sim} S_2\ H_2$. Thus proved.

B.2 Proof for Lemma 2

The proof is by structural induction on the derivation of $s\ \varphi_0\ S_1\ H_1 \Downarrow S_3\ H_3$. The rules [E-ASSIGN], [E-STORE] and [E-NEW] form the base cases. Rules [E-CALL], [E-TRUE], [E-FALSE] and [E-SEQ] form the inductive cases.

From the premise, we have:

$$S_1\ H_1 \overset{\triangleright\varphi}{\sim} S_2\ H_2 \tag{99}$$
$$\Gamma\ \Sigma \vdash S_1\ H_1 \tag{100}$$
$$\Gamma\ \Sigma \vdash S_2\ H_2 \tag{101}$$
$$\Gamma\ \Sigma \vdash s : \varphi_0 \tag{102}$$
$$s\ \varphi_0\ S_1\ H_1 \Downarrow S_3\ H_3 \tag{103}$$
$$s\ \varphi_0\ S_2\ H_2 \Downarrow S_4\ H_4 \tag{104}$$

Case [E-ASSIGN] Suppose the statement s is $x = y.f$. From the operational semantics, we know that only the stack is modified. This means that we have $H_3 = H_1$ and $H_4 = H_2$. S_1 and S_2 are the same as S_3 and S_4 respectively, except for the value of x. Let $S_1(y) = l_1\ \alpha_1$, $S_2(y) = l_2\ \alpha_2$ and $H_1(l_1)(f) = v_1\ \beta_1$, $H_2(l_2)(f) = v_2\ \beta_2$. From the operational semantics we have $S_3(x) = v_1\ \varphi_1$ and $S_4(x) = v_2\ \varphi_2$ where $\varphi_1 = (\varphi_0 \sqcup \alpha_1 \sqcup \beta_1 \sqcup \mathit{label}(x))$ and $\varphi_2 = (\varphi_0 \sqcup \alpha_2 \sqcup \beta_2 \sqcup \mathit{label}(x))$.

Suppose $\varphi_1 \sqcup \varphi_2 \trianglerighteq \varphi$. From Lemma 1, we have $\alpha_1 \trianglerighteq \varphi$, $\alpha_2 \trianglerighteq \varphi$, $\beta_1 \trianglerighteq \varphi$ and $\beta_2 \trianglerighteq \varphi$. Then by applying (99) we have $v_1 = v_2$, $\beta_1 = \beta_2$, $l_1 = l_2$ and $\alpha_1 = \alpha_2$. Hence $S_3(x) = S_4(x)$. Thus $S_3\ H_3 \overset{\triangleright\varphi}{\sim} S_4\ H_4$.

The arguments for $x = y$ run on similar lines.

Case [E-STORE] Suppose the statement is $x.f = y$. From the operational semantics, we know that only the heap is changed. This means that $S_3 = S_1$ and $S_4 = S_2$. Let $S_1(x) = l_1\ \alpha_1$, $S_2(x) = l_2\ \alpha_2$, $S_1(y) = v_1\ \beta_1$ and $S_2(y) = v_2\ \beta_2$. From operational semantics, we have H_3 and H_4 to be the same as H_1 and H_2 respectively, except that $H_3(l_1)(f) = v_1\ \varphi_1$ and $H_4(l_2)(f) = v_2\ \varphi_2$ where $\varphi_1 = (\varphi_0 \sqcup \alpha_1 \sqcup \beta_1 \sqcup \mathit{label}(f))$ and $\varphi_2 = (\varphi_0 \sqcup \alpha_2 \sqcup \beta_2 \sqcup \mathit{label}(f))$. Suppose $\varphi_1 \sqcup \varphi_2 \trianglerighteq \varphi$. From Lemma 1, we have $\alpha_1 \trianglerighteq \varphi$, $\alpha_2 \trianglerighteq \varphi$, $\beta_1 \trianglerighteq \varphi$ and $\beta_2 \trianglerighteq \varphi$. Then by applying (99) we have $v_1 = v_2$, $\beta_1 = \beta_2$, $l_1 = l_2$ and $\alpha_1 = \alpha_2$. Hence $H_3 = H_4$. Thus $S_3\ H_3 \overset{\triangleright\varphi}{\sim} S_4\ H_4$.

Case [E-NEW] Suppose the statement s is $x = \mathsf{new}^o\ c$. From the operational semantics, we know that H_3 and H_4 are the same as H_1 and H_2 respectively, except for a new introduced location. From [E-NEW], a value of the form $(\mathsf{null}\ label(f))$ are assigned to every field f of the added location in H_3 and H_4. Also S_1 and S_2 are the same as S_3 and S_4 respectively, except for a new introduced variable x. From [E-NEW] $S_3(x) = l\ (\varphi_0 \sqcup label(x))$ and $S_4 = l'\ (\varphi_0 \sqcup label(x))$. Owing to the equivalence due to renaming of locations, we have $S_3(x) = S_4(x)$ and $H_3(l)(f) = H_4(l')(f)$ for every field f. Thus $S_3\ H_3 \overset{\triangleright\varphi}{\sim} S_4\ H_4$.

Case [E-SEQ] Suppose the statement s is $s_1; s_2$. From (103), (104) and the operational semantics we have:

$$s_1\ \varphi_0\ E_1 \Downarrow E_1', \tag{105}$$

$$s_2\ \varphi_0\ E_1' \Downarrow E_3, \tag{106}$$

$$s_1\ \varphi_0\ E_2 \Downarrow E_2' \text{ and} \tag{107}$$

$$s_2\ \varphi_0\ E_2' \Downarrow E_4. \tag{108}$$

From static semantics we have:

$$A \vdash s_1 : \varphi_0 \text{ and} \tag{109}$$

$$A \vdash s_2 : \varphi_0. \tag{110}$$

Applying induction hypothesis on (99), (100), (101), (109), (105) and (107), we have:

$$E_1' \overset{\triangleright\varphi}{\sim} E_2'. \tag{111}$$

Applying Theorem 1 on (100), (109), (105), and on (101), (110), (106) we have:

$$A \vdash E_1' \text{ and} \tag{112}$$

$$A \vdash E_2'. \tag{113}$$

Now applying induction hypothesis on (111), (112), (113), (110), (106) and (108), we have $E_3 \overset{\triangleright\varphi}{\sim} E_4$. Hence proved.

Case [E-TRUE] Suppose the statement is $\mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$. Let $S_1(x) = l_1\ \alpha_1$ and $S_2(x) = l_2\ \alpha_2$.

Suppose $\alpha_1 \sqcup \alpha_2 \trianglerighteq \varphi$. From (99), we have $l_1 = l_2$ and $\alpha_1 = \alpha_2$. Consider the case when $l_1 \neq \mathsf{null}$ (same as $l_2 \neq \mathsf{null}$). The case for $l_1 = \mathsf{null}$ (same as $l_2 = \mathsf{null}$) follows on the same lines. Then from operational semantics we have:

$$s_1\ \varphi_0 \sqcup \alpha_1\ E_1 \Downarrow E_3 \text{ and} \tag{114}$$

$$s_1\ \varphi_0 \sqcup \alpha_2\ E_2 \Downarrow E_4. \tag{115}$$

Let $\Gamma(x) = \tau\ \gamma$. From (100), (101) and the static semantics we have $A \vdash s_1 : \varphi_0 \sqcup \gamma$, where $\alpha_1 \sqsubseteq \gamma$ and $\alpha_2 \sqsubseteq \gamma$. Applying [SUB-STM], we have:

$$A \vdash s_1 : \varphi_0 \sqcup \alpha_1 \text{ and} \tag{116}$$

$$A \vdash s_1 : \varphi_0 \sqcup \alpha_2. \tag{117}$$

Now since $\alpha_1 = \alpha_2$, applying induction hypothesis on (99), (100), (101), (114), (115) and (122), we have $E_3 \overset{\triangleright\varphi}{\sim} E_4$.

Suppose $\alpha_1 \sqcup \alpha_2 \not\trianglerighteq \varphi$. Then from Lemma 1 we have $\alpha_1 \not\trianglerighteq \varphi$ and $\alpha_2 \not\trianglerighteq$. This implies that:

$$\varphi_0 \sqcup \alpha_1 \not\trianglerighteq \varphi, \tag{118}$$

$$\varphi_0 \sqcup \alpha_2 \not\trianglerighteq \varphi. \tag{119}$$

Consider the case when $l_1 \neq \mathsf{null}$ and $l_2 = \mathsf{null}$. The other cases follow on the same lines. We have:

$$s_1\ \varphi_0 \sqcup \alpha_1\ E_1 \Downarrow E_3 \text{ and} \tag{120}$$

$$s_2\ \varphi_0 \sqcup \alpha_2\ E_2 \Downarrow E_4. \tag{121}$$

Let $\Gamma(x) = \tau\ \gamma$. From (100), (101) and the static semantics we have $A \vdash s_1 : \varphi_0 \sqcup \gamma$ and $A \vdash s_2 : \varphi_0 \sqcup \gamma$, where $\alpha_1 \sqsubseteq \gamma$ and $\alpha_2 \sqsubseteq \gamma$. Applying [SUB-STM], we have:

$$A \vdash s_1 : \varphi_0 \sqcup \alpha_1 \text{ and} \tag{122}$$

$$A \vdash s_2 : \varphi_0 \sqcup \alpha_2. \tag{123}$$

Applying Lemma 5 on (122), (100), (120), (118) and on (123), (101), (121), (119) we have:

$$E_1 \overset{\triangleright\varphi}{\sim} E_3, \tag{124}$$

$$E_2 \overset{\triangleright\varphi}{\sim} E_4. \tag{125}$$

From (99), (124), (125), we finally have $E_3 \overset{\triangleright\varphi}{\sim} E_4$.

Case [E-FALSE] Similar to the case of [E-TRUE].

Case [E-CALL] Suppose the statement s is $x.m(y)$. From the operational semantics, we know that $S_3 = S_1$ and $S_4 = S_2$. Let $S_1(x) = l_1^{o_1}\ \alpha_1$, $S_2(x) = l_2^{o_2}\ \alpha_2$, $S_1(y) = v_1\ \beta_1$, $S_2(y) = v_2\ \beta_2$, $\Gamma(x) = \tau_x\ \varphi_x$ and $\Gamma(y) = \tau_y\ \varphi_y$. From (100) and (101), we have $\{o_1, o_2\} \subseteq \tau_x$ and $\alpha_1 \sqsubseteq \varphi_x$, $\alpha_2 \sqsubseteq \varphi_x$, $\beta_1 \sqsubseteq \varphi_y$ and $\beta_2 \sqsubseteq \varphi_y$.
Let $S_1^0 = \{\mathsf{this} \mapsto (l_1^{o_1}\ \alpha_1), z \mapsto (v_1\ \beta_1 \sqcup \mathit{label}(z))\}$. Let $S_2^0 = \{\mathsf{this} \mapsto (l_2^{o_2}\ \alpha_2), z \mapsto (v_2\ \beta_2 \sqcup \mathit{label}(z))\}$. Let

$$\mathit{method}(\mathit{type}(o_1), m) = m(z)\{s_1\},$$

$$\mathit{method}(\mathit{type}(o_2), m) = m(z)\{s_2\}.$$

From operational semantics, we have:

$$s_1\ (\varphi_0 \sqcup \alpha_1)\ S_1^0\ H_1 \Downarrow S_3^0\ H_3 \text{ and} \tag{126}$$

$$s_2\ (\varphi_0 \sqcup \alpha_2)\ S_2^0\ H_2 \Downarrow S_4^0\ H_4. \tag{127}$$

Let $\Gamma_0 = \{\mathsf{this} \mapsto (\tau_x\ \varphi_x), z \mapsto (\tau_y\ \varphi_y)\}$. From [CALL] we have $\Gamma_0\ \Sigma \vdash s_i : \varphi_0 \sqcup \varphi_x$, $1 \leq i \leq 2$ and from [SUB-STM] of static semantics, we have:

$$\Gamma_0\ \Sigma \vdash s_1 : \varphi_0 \sqcup \alpha_1, \tag{128}$$

$$\Gamma_0\ \Sigma \vdash s_2 : \varphi_0 \sqcup \alpha_2. \tag{129}$$

From (100) and (101), we have:

$$\Gamma_0\ \Sigma \vdash S_1^0\ H_1, \tag{130}$$

$$\Gamma_0\ \Sigma \vdash S_2^0\ H_2. \tag{131}$$

From (99), we have:

$$S_1^0\ H_1 \overset{\triangleright\varphi}{\sim} S_2^0\ H_2. \tag{132}$$

Suppose $\alpha_1 \sqcup \alpha_2 \trianglerighteq \varphi$. Applying (99), we have that $l_1^{o_1} = l_2^{o_2}$, $\alpha_1 = \alpha_2$. This means that the same method is invoked and $s_1 = s_2$. Then applying induction hypothesis on (132), (130), (131), (128), (126), (127), we have $S_3^0\ H_3 \overset{\triangleright\varphi}{\sim} S_4^0\ H_4$. Because the stack is not changed by the statement $x.m(y)$, we have $S_3\ H_3 \overset{\triangleright\varphi}{\sim} S_4\ H_4$.

Suppose $\alpha_1 \sqcup \alpha_2 \not\trianglerighteq \varphi$. Then we have $\alpha_1 \not\trianglerighteq \varphi$ and $\alpha_2 \not\trianglerighteq \varphi$. From Lemma 1 we have:

$$\varphi_0 \sqcup \alpha_1 \not\trianglerighteq \varphi, \tag{133}$$

$$\varphi_0 \sqcup \alpha_2 \not\trianglerighteq \varphi. \tag{134}$$

Then applying Lemma 5 on (128), (130), (126), (133) and (129), (131), (127), (134) we have:

$$S_1^0\ H_1 \overset{\triangleright\varphi}{\sim} S_3^0\ H_3, \tag{135}$$

$$S_2^0\ H_2 \overset{\triangleright\varphi}{\sim} S_4^0\ H_4. \tag{136}$$

Then from (132), (135), (136), we have $S_3^0\ H_3 \overset{\triangleright\varphi}{\sim} S_4^0\ H_4$. Because the stack is not changed by the statement $x.m(y)$, we have $S_3\ H_3 \overset{\triangleright\varphi}{\sim} S_4\ H_4$. Hence proved.

B.3 Proof of Lemma 3

From the premise, we have:

$$S_1\ H_1 \overset{\rhd\varphi}{\approx} S_2\ H_2 \tag{137}$$

$$\vdash S_1\ H_1 \tag{138}$$

$$\vdash S_2\ H_2 \tag{139}$$

We only prove for the case of stack variables. The case of fields follows on the same lines. Let $S_1(x) = v_1\ \alpha_1$ and $S_2 = v_2\ \alpha_2$. Suppose $\alpha_1 \sqcup \trianglerighteq\varphi$. From (138) and (139), we have $label(x) \sqsubseteq \alpha_1 \sqcup \alpha_2$. Then from Lemma 1, we have that $label(x) \trianglerighteq \varphi$. Now from (137), we have that $S_1(x) = S_2(x)$. Hence $S_1\ H_1 \overset{\rhd\varphi}{\sim} S_2\ H_2$.

B.4 Proof of Lemma 4

From the premise, we have:

$$S_1\ H_1 \overset{\rhd\varphi}{\sim} S_2\ H_2 \tag{140}$$

$$\vdash S_1\ H_1 \tag{141}$$

$$\vdash S_2\ H_2 \tag{142}$$

We only prove for the case of stack variables. The case of fields follows on the same lines. Let $S_1(x) = v_1\ \alpha_1$ and $S_2 = v_2\ \alpha_2$. Ssuppose $label(x) = \varphi$. From (141) and (142), we have $\alpha_1 \trianglerighteq label(x)$ and $\alpha_2 \trianglerighteq label(x)$. This implies that $\alpha_1 \sqcup \alpha_2 \trianglerighteq \varphi$. Then from (140), we have $S_1(x) = S_2(x)$. Hence $S_1\ H_1 \overset{\varphi}{\approx} S_2\ H_2$.