Secure Program Partitioning

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers
Computer Science Department
Cornell University

October 18, 2001

Abstract

This paper presents secure program partitioning, a language-based technique for protecting confidential data during computation in distributed systems containing mutually untrusted hosts. Confidentiality and integrity policies can be expressed by annotating programs with security types that constrain information flow; these programs can then be partitioned automatically to run securely on heterogeneously trusted hosts. The resulting communicating subprograms collectively implement the original program, yet the system as a whole satisfies the security requirements of participating principals without requiring a universally trusted host machine. The experience in applying this methodology and the performance of the resulting distributed code suggest that this is a promising way to obtain secure distributed computation.

This Technical Report is an expanded version of the published paper "Untrusted Hosts and Confidentiality: Secure Program Partitioning" [51]. The main difference between the two is Appendix A, which contains a correctness proof for the control-transfer protocols described in Section 5.

1 Introduction

A significant challenge for computer systems, especially distributed systems, is maintaining the confidentiality and integrity of the data they manipulate. Existing techniques cannot ensure that an entire computing system satisfies a security policy for data confidentiality and integrity.[1] Standard mechanisms, such as access control and encryption, are essential tools for ensuring that system components do not violate these security policies. However, for systems that contain non-trivial computational components, access control and encryption are much less helpful for ensuring (and proving) that the system obeys the desired security policies.

A requirement that controls the *end-to-end* use of data in a secure system is an *information-flow policy* [3, 4, 7, 8, 15]. Information-flow policies are the natural way to specify confidentiality and integrity requirements because these policies constrain how information is used by the entire system, rather than simply regulating which principals (users, machines, programs, or other entities) can read or modify the data at particular points during execution. An informal example of such a confidentiality policy is "the information contained in my bank account file may be obtained only by me and the bank managers." Because it controls information rather than access, this policy is considerably stronger than the similar access control policy, "only processes authorized by me or bank managers

Cornell University Computer Science Department Technical Report 2001–1846.

This research was supported in part by DARPA Contracts F30602-98-1-0237 and F30602-99-1-0533, monitored by USAF Rome Laboratory, and in part by ONR Grant N00014-01-1-0968. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

[1]*Confidentiality* is used here as a synonym for secrecy; it is an important aspect of privacy.

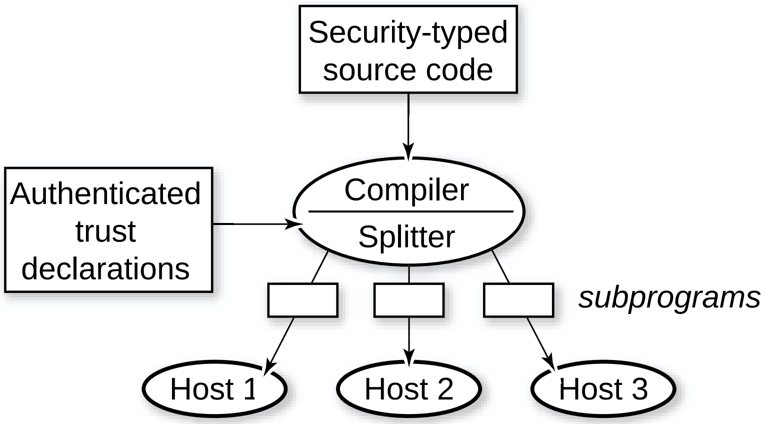

Figure 1: Secure program partitioning

may open the file containing my bank account information." This paper addresses the problem of how to practically specify and enforce information-flow policies in distributed systems.

A promising approach for describing such policies is the use of *security-typed languages* [1, 17, 27, 35, 42, 46, 50]. In this approach, explicit program annotations specify restrictions on the flow of information, and the language implementation (the compiler and run-time system) rejects programs that violate the restrictions. The program does not have to be trusted to enforce the security policy; only the compiler must be trusted. Static analysis also offers advantages over run-time enforcement because any purely run-time mechanism can enforce only safety properties, which excludes many useful information-flow policies [40].

To date, security-typed languages have addressed information-flow security in systems executed on a single trusted host. This assumption is unrealistic, particularly in scenarios for which information-flow policies are most desirable—when multiple principals need to cooperate but do not entirely trust one another. Simple examples of such scenarios abound: email services, web-based shopping and financial planning, business-to-business transactions, and joint military information systems. We expect sophisticated, collaborative, inter-organizational computation to become increasingly common; some way is needed to assure that data confidentiality is protected.

The general problem with these collaborative computations is ensuring that the security policies of all the participants are enforced. When participants do not fully trust each others' hosts, it is necessary to distribute the data and computational work among the hosts. This distribution creates a new threat to security: the hosts used for computation may cause security violations—either directly, by leaking information, or indirectly, by carrying out computations in a way that causes other hosts to leak information. Of course, the program itself may also cause security violations. Because existing single-host techniques address this problem, we focus on the new threat, untrusted hosts.

In this paper, we present *secure program partitioning*, a novel way to protect the confidentiality of data for computations that manipulate data with differing confidentiality needs on an execution platform comprising heterogeneously trusted hosts. Figure 1 illustrates the key insight: The security policy can be used to guide the automatic splitting of a security-typed program into communicating subprograms, each running on a different host. Collectively, the subprograms perform the same computation as the original; in addition, they satisfy all the participants' security policies without requiring a single universally trusted host. We are primarily interested in enforcing confidentiality policies; in this setting, however, enforcement of confidentiality requires enforcement of simple integrity policies as well.

The splitter receives two inputs: the program, including its confidentiality and integrity policy annotations, and also a set of signed trust declarations stating each principal's trust in hosts and other principals. The goal of secure program partitioning is to ensure that if a host h is subverted, the only data whose confidentiality or integrity is threatened is data owned by principals that have declared they trust h.

It is useful to contrast this approach with the usual development of secure distributed systems, which involves the careful design of protocols for exchanging data among hosts in the system. By contrast, our approach provides the following benefits:

- **Stronger security:** Secure program partitioning can be applied to information-flow policies; most distributed systems make no attempt to control information flow. It can also be applied to access control policies, which are comparatively simple to enforce with this technique.
- **Decentralization:** Collaborative computations can be carried out despite incomplete trust. In addition, for many computations, there is no need for a universally trusted host. Each participant can independently ensure that its security policies are enforced.
- **Automation:** Large computing systems with many participating parties contain complex, interacting security policies that evolve over time; automated enforcement is becoming a necessity. Secure program partitioning permits a computation to be described as a single program independent of its distributed implementation. The partitioning process then *automatically* generates a secure protocol for data exchange among the hosts.
- **Explicit policies:** Security-typed programs force policy decisions to be made explicit in the system design, making them auditable and automatically verifiable. Type checking can then reveal subtle design flaws that make security violations possible.

Secure program partitioning has the most value when strong protection of confidentiality is needed by one or more principals, the computing platform consists of differently trusted hosts, there is a generally agreed-upon computation to be performed, and security, performance, or functionality considerations prevent the entire computation from being executed on a single host. One example of a possible application is an integrated medical information system that stores patient and physician records, raw test data, and employee records, and supports information exchange with other medical institutions. Another example is an automated business-to-business procurement system, in which profitable negotiation by the buyer and supplier depends on keeping some data confidential.

This paper describes Jif/split, our implementation of secure program partitioning, which includes a static checker, program splitter, and run-time support for the distributed subprograms. We present simple examples of applying this approach and give performance results that indicate its practicality.

Our system can express security policies that control covert and overt storage channels. However, certain classes of information-flow policies are not controlled by our system: timing and termination channels, which would be more important in a malicious-code setting. Language-based work on timing and termination flows, largely orthogonal to this work, is ongoing elsewhere (e.g., [2, 42]).

The rest of the paper is structured as follows. The next section describes the model for writing secure programs in a Java-like language that permits the specification of information-flow policies. Section 3 describes the assumptions about the networked environment, and discusses the assurance that secure program partitioning can provide in this environment. Section 4 describes the static conditions that are imposed when a program is split, including additional static checks needed in a distributed environment. Section 5 covers the dynamic (run-time) checks that are needed in addition to prevent attackers from violating the assumptions of the static checking. Section 6 describes the partitioning translation, including the optimization techniques for arriving at efficient split programs. Section 7 gives details of our prototype implementation and reports performance results. Section 8 discusses the trusted computing base and shows that it can be made small and localized to trusted hosts. Related and future work is considered in Sections 9 and 10. Section 11 concludes.

2 Secure Programming Model

The Jif/split program splitter extends the compiler for Jif [27, 29], a security-typed extension to Java that incorporates confidentiality labels from the decentralized label model [28]. In this model, principals can express ownership in data; the correctness of secure partitioning is defined in terms of this idea of ownership. The label model supports selective declassification, a feature needed for realistic applications of information-flow control.

2.1 Security Labels

Central to the model is the notion of a *principal*, which is an entity (e.g., user, process, party) that can have a confidentiality or integrity concern with respect to data. Principals can be named in information-flow policies and are also used to define the *authority* possessed by the running program. The authority at a point in the program is simply a set of principals that are assumed to authorize any action taken by the program at that point. Different program points may have different authority, which must be explicitly granted by the principals in question.

Security *labels* express confidentiality policies on data in a program; they provide the core vocabulary of the overall system security policy. A simple label is written $\{\mathtt{o}\mathtt{:}\mathtt{r}_1\mathtt{,}\mathtt{r}_2\mathtt{,}\ldots\mathtt{,}\mathtt{r}_n\}$, meaning that the labeled data is owned by principal `o`, and that `o` permits the data to be read by principals $\mathtt{r}_1$ through $\mathtt{r}_n$ (and, implicitly, `o`).

Data may have multiple owners, each controlling a different component of its label. For example, the label $\{\mathtt{o}_1\mathtt{:}\mathtt{r}_1\mathtt{,}\mathtt{r}_2\mathtt{;}\ \mathtt{o}_2\mathtt{:}\mathtt{r}_1\mathtt{,}\mathtt{r}_3\}$ contains two components and says that owner $\mathtt{o}_1$ allows readers $\mathtt{r}_1$ and $\mathtt{r}_2$ and owner $\mathtt{o}_2$ allows readers $\mathtt{r}_1$ and $\mathtt{r}_3$. Because *all* of the policies described by a label must be obeyed, only $\mathtt{r}_1$ will be able to read data with this annotation. Such composite labels arise naturally in collaborative computations: for example, if `x` has label $\{\mathtt{o}_1\mathtt{:}\mathtt{r}_1\mathtt{,}\mathtt{r}_2\}$ and `y` has label $\{\mathtt{o}_2\mathtt{:}\mathtt{r}_1\mathtt{,}\mathtt{r}_3\}$, then the sum `x + y` has the composite label $\mathtt{int}\{\mathtt{o}_1\mathtt{:}\mathtt{r}_1\mathtt{,}\mathtt{r}_2\mathtt{;}\ \mathtt{o}_2\mathtt{:}\mathtt{r}_1\mathtt{,}\mathtt{r}_3\}$, which expresses the conservative requirement that the sum is subject to both the policy on `x` and the policy on `y`.

In this paper, the decentralized label model is extended with label components that specify integrity. The label $\{\mathtt{?:}\mathtt{p}_1\mathtt{,}\ldots\mathtt{,}\mathtt{p}_n\}$ specifies that principals $\mathtt{p}_1$ through $\mathtt{p}_n$ *trust* the data—they believe the data to be computed by the program as written. (Because integrity policies have no owner, a question mark is used in its place.) This is a weak notion of trust; its purpose is to protect security-critical information from damage by subverted hosts. Labels combining integrity and confidentiality components also arise naturally.

We write $\mathtt{L}_1 \sqsubseteq \mathtt{L}_2$ if the label $\mathtt{L}_1$ is less restrictive than the label $\mathtt{L}_2$. Intuitively, data with label $\mathtt{L}_1$ is less confidential than data with label $\mathtt{L}_2$—more principals are permitted to see the data, and, consequently, there are fewer restrictions on how data with label $\mathtt{L}_1$ may be used. For example, $\{\mathtt{o}\mathtt{:}\mathtt{r}\} \sqsubseteq \{\mathtt{o}\mathtt{:}\}$ holds because the left label allows both `o` and `r` to read the data, whereas the right label admits only `o` as a reader.

The relation $\sqsubseteq$ is a pre-order whose equivalence classes form a distributive lattice; we write $\sqcup$ and $\sqcap$ for the lattice join and meet operations, respectively. The label join operation combines the restrictions on how data may be used. As in the example above, if `x` has label $\mathtt{L}_1$ and `y` has label $\mathtt{L}_2$, the sum `x + y` has label $\mathtt{L}_1 \sqcup \mathtt{L}_2$, which includes the restrictions of both.

For any label `L`, the functions $C(\mathtt{L})$ and $I(\mathtt{L})$ extract the confidentiality and integrity parts of `L`, respectively. Because confidentiality and integrity are duals [4], if $\mathtt{L}_1 \sqsubseteq \mathtt{L}_2$, then $\mathtt{L}_2$ must specify at least as much confidentiality and at *most* as much integrity as $\mathtt{L}_1$. This interpretation is consistent with the idea that labels represent restrictions on how data may be used; data with higher integrity has *fewer* restrictions on its use.

Types in Jif are labeled, allowing the programmer to declare variables and fields that include security annotations. For example, a value with type $\mathtt{int}\{\mathtt{o}\mathtt{:}\mathtt{r}\}$ is an integer owned by principal `o` and readable by `r`. When unlabeled Java types are written in a program, the label component is automatically inferred.

Every program expression has a labeled type that indicates an upper bound (with respect to the $\sqsubseteq$ order) of the security of the data represented by the expression. Jif's type-checking algorithm prevents labeled information from being *downgraded*, or assigned a less-restrictive label (i.e., lower in the lattice). In general, downgrading results in a loss of confidentiality or a spurious increase in claimed integrity. The type system tracks data dependencies (information flows) to prevent unintentional downgrading.

2.2 Declassification

Systems for enforcing information-flow policies have often run into practical difficulties. In part this has resulted from their basis in the security property of *noninterference* [15], which captures the requirement that data labeled `L` cannot affect any data whose label is not at least as restrictive. Noninterference allows the expression of controls on the end-to-end information flow within a system, but it does not provide sufficient expressive power: realistic systems require limited violations of noninterference, such as the ability to release encrypted data. An important feature of the decentralized label model is the ability to write computations that include controlled forms of downgrading, providing an escape hatch from strict noninterference.

Downgrading confidentiality is called *declassification*; it is provided in Jif by `declassify(e, L)`, which allows a program acting with sufficient authority to declassify the expression `e` to label `L`. A principal p's authority is needed to perform declassifications of data owned by p. For example, owner `o` can add a reader `r` to a piece of data by declassifying its label from $\{$`o:`$\}$ to $\{$`o:r`$\}$.

The integrity counterpart to `declassify` is `endorse`, which allows a principal to declare trust in a piece of data based on information outside the program text. For example, a principal might `endorse` a message after verifying that it has been signed by a trusted principal. Neither `declassify` nor `endorse` has a run-time cost; they simply change the label of the security type of their argument.

2.3 Implicit Flows

One complication for security-typed languages is *implicit flows*, which arise from the control flow of the program. Consider this example in which four program points (A–D) are indicated by arrows:

$$\uparrow_A \texttt{if x then } \uparrow_B \texttt{y = true; else } \uparrow_C \texttt{y = false;} \uparrow_D$$

This code creates a dependency between the value `x`, which has type `boolean{L}`, and the value stored in `y`—the code is equivalent to the assignment `y = x`. For this assignment to be secure, `y`'s label must be at least as restrictive as `L`. Note that in the example information flows from `x` to `y` even though only constant values are assigned to `y`.

To control these implicit information flows, a label is assigned to each program point, indicated by the arrows. From a confidentiality standpoint, the label captures the information that can be learned by knowing that the program reached that point during execution; from an integrity standpoint, it captures the integrity of the information that determines the control flow to that point. In this example, if the label of program point $\uparrow_A$ is $\mathtt{L}_A$, the label at point $\uparrow_B$ is $\mathtt{L}_A \sqcup \mathtt{L}$ because reaching point $\uparrow_B$ depends on both reaching point $\uparrow_A$ and the value of `x`, which has label `L`. Similarly, $\uparrow_C$ also has label $\mathtt{L}_A \sqcup \mathtt{L}$. Reaching point $\uparrow_D$ depends only on reaching point $\uparrow_A$ (both branches fall through to point $\uparrow_D$), so it has label $\mathtt{L}_A$.

Because naming program points is quite cumbersome, we introduce a special label, $\underline{\mathtt{pc}}$, which is the label of the program counter at each program point. Which program point $\underline{\mathtt{pc}}$ refers to is usually clear from context, so we might say "the $\underline{\mathtt{pc}}$ inside the branch is $\mathtt{L}_A \sqcup \mathtt{L}$." To conservatively control implicit flows, the label for any expression in the program includes the $\underline{\mathtt{pc}}$ label for that program point. For example, it means that the assignment `y = true` is allowed only if `y`'s label is at least as restrictive as $\mathtt{L}_A \sqcup \mathtt{L}$, which correctly captures `y`'s dependency on `x`.

Using the labels provided by the programmer and the inferred $\underline{\mathtt{pc}}$ label, the compiler is able to statically verify that all of the information flows apparent in the program text satisfy the label constraints that prevent illegal information flows from occurring. If the program does not satisfy the security policy, it is rejected.

2.4 Language Features

In addition to these changes to the Java type system, Jif adds a number of constructs for creating secure programs. The following are germane to this paper:

```
public class OTExample {
  int{Alice:; ?:Alice} m1;
  int{Alice:; ?:Alice} m2;
  boolean{Alice:; ?:Alice} isAccessed;

  int{Bob:} transfer{?:Alice} (int{Bob:} n)
  where authority(Alice) {
    int tmp1 = m1;
    int tmp2 = m2;
    if (!isAccessed) {
      isAccessed = true;
      if (endorse(n, {?:Alice}) == 1)
        return declassify(tmp1, {Bob:});
      else
        return declassify(tmp2, {Bob:});
    }
    else return 0;
  }
}
```

Figure 2: Oblivious transfer code

- An optional `authority` clause on method declarations describes the authority available in the body of the method. Code containing such a clause can be added to the system only with the permission of the principals named in it.
- Optional label bounds on the initial and final $\underline{pc}$ labels of a method. For example, the method signature

  $$\texttt{int}\{\texttt{L}_1\}\ \texttt{m}\{\texttt{I}\}(\texttt{int}\{\texttt{L}_2\}\ \texttt{x})\texttt{:}\ \{\texttt{F}\}$$

  means that the method `m` can only be called when $\underline{pc} \sqsubseteq \texttt{I}$. It takes an integer `x` with label $\texttt{L}_2$ and returns an integer labeled $\texttt{L}_1$. Upon exiting `m`, the condition $\underline{pc} \sqsubseteq \texttt{F}$ holds.

Jif also introduces some limitations to Java, which apply to this work as well. The most important is that programs are assumed to be sequential: the `Thread` class is not available. This limitation prevents an important class of timing channels whose control is an open research area. Providing support for full-fledged threaded and concurrent distributed programming is the focus of ongoing work [22, 41, 42].

2.5 Oblivious Transfer Example

Figure 2 shows a sample program that we will use as a running example. It is based on the well-known Oblivious Transfer Problem [11, 36], in which the principal Alice has two values (here represented by fields `m1` and `m2`), and Bob may request exactly one of the two values. However, Bob does not want Alice to learn which of the two values was requested. We chose this example because it is short, has interesting security issues, and has been well studied: for instance, it is known that a trusted third party is needed for a secure distributed implementation [6].[2]

Alice's secret data is represented by fields `m1` and `m2`. The policy {`Alice:; ?:Alice`} indicates that these fields are owned by Alice, that she lets no one else read them, and that she trusts their contents. The boolean `isAccessed` records whether Bob has requested a value yet.

[2]Probabilistic solutions using two hosts exist, but these algorithms leak small amounts of information. Because Jif's type system is geared to possibilistic information flows, these probabilistic algorithms are rejected as potentially insecure. Ongoing research [16, 45, 39] attempts to address probabilistic security.

Lines 6 through 18 define a method `transfer` that encapsulates the oblivious transfer protocol. It takes a request, `n`, owned by Bob, and returns either `m1` or `m2` depending on `n`'s value. Note that because Alice owns `m1` and `m2`, releasing the data requires declassification (lines 13 and 15). Her authority, needed to perform this declassification, is granted by the `authority` clause on line 7.

Ignoring for now the temporary variables `tmp1` and `tmp2` and the `endorse` statement, the body of the `transfer` method is straightforward: Line 10 checks whether Bob has made a request already. If not, line 11 records the request, and lines 12 through 15 return the appropriate field after declassifying them to be visible by Bob. If Bob has already made a request, `transfer` simply returns `0`.

The simplicity of this program is deceptive. For example, the $\underline{\texttt{pc}}$ label at the start of the `transfer` method must be bounded above by the label $\{\texttt{?:Alice}\}$, as indicated on line 6. The reason is that line 11 assigns `true` into the field `isAccessed`, which requires Alice's integrity. If the program counter at the point of assignment does not also have Alice's trust, the integrity of `isAccessed` is compromised.

Other subtle interactions between confidentiality, integrity, and trust explain the need for the temporary variables and endorsement. We shall discuss these interactions throughout the rest of the paper as we describe security considerations in a distributed environment. One benefit of programming in a security-typed language is that the compiler can catch many subtle security holes even though the code is written in a style that contains no specification of how the code is to be distributed.

3 Assumptions and Assurance

The goal of secure program partitioning is to take a security-typed source program and a description of trust relationships and (if possible) produce a distributed version of the same program that executes securely in any consistent environment. This section discusses our assumptions about the distributed environment and describes the confidentiality and integrity assurance that can be provided in this environment.

3.1 Target environment

Clearly, any secure distributed system relies on the trustworthiness of the underlying network infrastructure. Let H be a set of *known hosts*, among which the program is to be distributed. We assume that pairwise communication between two members of H is reliable, in-order, and cannot be intercepted by hosts outside H or by the other members of H. Protection against interception can be achieved efficiently through well-known encryption techniques (e.g, [43, 48]); for example, each pair of hosts can use symmetric encryption to exchange information, with key exchange via public-key encryption. We assume that the same encryption mechanisms permit each member of H to authenticate messages sent and received by one another.

To securely partition a program, the splitter must know the trust relationships between the participating principals and the hosts H. To capture this information, we need two pieces of data about each host h:

- A *confidentiality* label $\texttt{C}_h$ that describes an upper bound on the confidentiality of information that can be sent securely to host h.
- An *integrity* label $\texttt{I}_h$ describing which principals trust data received from h.

These trust declarations are public knowledge—that is, they are available on all known hosts—and are signed by the principals involved. We assume the existence of a public-key infrastructure that makes digital signatures feasible.

Consider a host A owned by Alice but untrusted by Bob, and a host B owned by Bob and untrusted by Alice. A reasonable trust model might be:

$$\begin{array}{ll} \texttt{C}_A = \{\texttt{Alice:}\} & \texttt{I}_A = \{\texttt{?:Alice}\} \\ \texttt{C}_B = \{\texttt{Bob:}\} & \texttt{I}_B = \{\texttt{?:Bob}\} \end{array}$$

Because Bob does not appear as an owner in the label $\mathtt{C}_A$, this description acknowledges that Bob is unwilling to send his private data to host A. Similarly, Bob does not trust information received from A because Bob does not appear in $\mathtt{I}_A$. The situation is symmetric with respect to Alice and Bob's host.

Next, consider hosts T and S that are partially trusted by Alice and Bob:

$$\mathtt{C}_T = \{\mathtt{Alice:;Bob:}\} \quad \mathtt{I}_T = \{\mathtt{?:Alice}\}$$
$$\mathtt{C}_S = \{\mathtt{Alice:;Bob:}\} \quad \mathtt{I}_S = \{\mathtt{?:}\}$$

Alice and Bob both trust T not to divulge their data incorrectly; on the other hand, Bob believes that T may corrupt data—he does not trust the integrity of data received from T. Host S is also trusted with confidential data, but neither Alice nor Bob trust data generated by S.

We will use hosts A, B, T, and S when discussing various partitions of the oblivious transfer algorithm in what follows.

3.2 Security assurance

Our goal is to ensure that the threats to a principal's confidential data are not increased by the failure or subversion of an untrusted host that is being used for execution. Bad hosts—hosts that fail or are subverted—have full access to the part of the program executing on them, can freely fabricate apparently authentic messages from bad hosts, and can share information with other bad hosts. Bad hosts may execute concurrently with good hosts, whereas good hosts preserve the sequential execution of the source language—there is only one good host executing at a time. However, we assume that bad hosts are not able to forge messages from good hosts, nor can they generate certain capabilities to be described later.

It is important to distinguish between intentional and unintentional release of confidential information. It is assumed that the `declassify` expressions in the original program intentionally release confidential data—that the principal authorizing that declassification trusts the program logic controlling its use. However, bad hosts should not be able to subvert this logic and cause more data to be released than intended. In programs with no `declassify` expressions, the failure or subversion of an untrusted host should not cause data to be leaked.

The security of a principal is endangered only if one or more of the hosts that the principal trusts is bad. Suppose the host h is bad and let $\mathtt{L}_e$ be the label of an expression in the program. The confidentiality of the expression's value is endangered only if $C(\mathtt{L}_e) \sqsubseteq \mathtt{C}_h$; correspondingly, the expression's integrity may have been corrupted only if $\mathtt{I}_h \sqsubseteq I(\mathtt{L}_e)$.

If Alice's machine A from Section 3.1 is compromised, only data owned by Alice may be leaked, and only data she trusts may be corrupted. Bob's privacy and integrity are protected. By contrast, if the semi-trusted machine T malfunctions, Alice and Bob's data may be leaked, but only Alice's data may be corrupted because only she trusts the integrity of the machine.

If there are multiple bad machines, they may cooperate to leak or corrupt more data. Our system is intended to enforce the following property:

> **Security Assurance:** The confidentiality of an expression e is not threatened by a set H_{bad} of bad hosts unless $C(\mathtt{L}_e) \sqsubseteq \bigsqcup_{h \in H_{\text{bad}}} \mathtt{C}_h$; its integrity is not threatened unless $\bigsqcap_{h \in H_{\text{bad}}} \mathtt{I}_h \sqsubseteq I(\mathtt{L}_e)$.

Providing this level of assurance involves two challenges: (1) Data with a confidentiality label (strictly) higher than $\mathtt{C}_h$ should never be sent (explicitly or implicitly) to h, and data with an integrity label lower than $\mathtt{I}_h$ should never be accepted from h. (2) Bad hosts should not be able to exploit the downgrading abilities of more privileged hosts, causing them to violate the security policy of the source program. The next two sections describe how a combination of static and dynamic mechanisms achieves this goal.

4 Static Security Constraints

At a high level, the partitioning process can be seen as a constraint satisfaction problem. Given a source program and the trust relationships between principals and hosts, the splitter must assign a host in H to each field, method, and program statement in the program. This fine-grained partitioning of the code is important so that a single method may access data of differing confidentiality and integrity. The primary concern when assigning hosts is to enforce the confidentiality and integrity requirements on data; efficiency, discussed in Section 6, is secondary. This section describes the static constraints on host selection.

4.1 Field and Statement Host Selection

Consider the field `m1` of the oblivious transfer example. It has label $\{\texttt{Alice:;\ ?:Alice}\}$, which says that Alice owns and trusts this data. Only certain hosts are suitable to store this field: hosts that Alice trusts to protect both her confidentiality and integrity. If the field were stored elsewhere, the untrusted host could violate Alice's policy, contradicting the security assurance of Section 3.2. The host requirements can be expressed using labels: $\{\texttt{Alice:}\} \sqsubseteq \texttt{C}_h$ and $\texttt{I}_h \sqsubseteq \{\texttt{?:Alice}\}$. The first inequality says that Alice allows her data to flow to h, and the second says that Alice trusts the data she receives from h. In general, for a field f with label $\texttt{L}_f$ we require

$$C(\texttt{L}_f) \sqsubseteq \texttt{C}_h \quad \text{and} \quad \texttt{I}_h \sqsubseteq I(\texttt{L}_f).$$

This same reasoning further generalizes to the constraints for locating an arbitrary program statement, S. Let $U(S)$ be the set of values *used* in the computation of S and let $D(S)$ be the set of locations S *defines*. Suppose that the label of the value v is $\texttt{L}_v$ and that the label of a location l is $\texttt{L}_l$. Let

$$\texttt{L}_{\text{in}} = \bigsqcup_{v \in U(S)} \texttt{L}_v \quad \text{and} \quad \texttt{L}_{\text{out}} = \bigsqcap_{l \in D(S)} \texttt{L}_l$$

A host h can execute the statement S securely, subject to constraints similar to those for fields.

$$C(\texttt{L}_{\text{in}}) \sqsubseteq \texttt{C}_h \quad \text{and} \quad \texttt{I}_h \sqsubseteq I(\texttt{L}_{\text{out}})$$

4.2 Preventing Read Channels

The rules for host selection for fields in the previous section are necessary but not sufficient in the distributed environment. Because bad hosts in the running system may be able to observe read requests from good hosts, a new kind of implicit flow is introduced: a *read channel* in which the request to read a field from a remote host itself communicates information.

For example, a naive implementation of the oblivious transfer example of Figure 2 exhibits a read channel. Suppose that in implementing the method `transfer`, the `declassify` expressions on lines 13 and 15 directly declassified the fields `m1` and `m2`, respectively, instead of the variables `tmp1` and `tmp2`. According to Bob, the value of the variable `n` is private and not to be revealed to Alice. However, if `m1` and `m2` are stored on Alice's machine, Alice can improperly learn the value of `n` from the read request.

The problem is that Alice can use read requests to reason about the location of the program counter. Therefore, the program counter at the point of a read operation must not contain information that the field's host is not allowed to see. With each field f, the static checker associates a confidentiality label $\texttt{Loc}_f$ that bounds the security level of implicit flows at each point where f is read. For each read of the field f, the label $\texttt{Loc}_f$ must satisfy the constraint $C(\underline{\texttt{pc}}) \sqsubseteq \texttt{Loc}_f$. Using this label $\texttt{Loc}_f$, the confidentiality constraint on host selection for the field is:

$$C(\texttt{L}_f) \sqcup \texttt{Loc}_f \sqsubseteq \texttt{C}_h$$

To eliminate the read channel in the example while preventing Bob from seeing both `m1` and `m2`, a trusted third party is needed. The programmer discovers this problem during development when the naive approach fails to split in a configuration with just the hosts A and B as described in Section 3.1. The error pinpoints the read channel introduced: arriving at line 13 depends on the value of `n`, so performing a request for `m1` there leaks `n` to Alice. The splitter automatically detects this problem when the field constraint above is checked.

If the more trusted host T is added to the set of known hosts, the splitter is able to solve the problem, even with the naive code, by allocating `m1` and `m2` on T, which prevents Alice from observing the read request. If S is used in place of T, the naive code again fails to split—even though S has enough privacy to hold Alice's data, fields `m1` and `m2` can't be located there because Alice doesn't trust S not to corrupt her data. Again, the programmer is warned of the read channel, but this time a different solution is possible: adding `tmp1` and `tmp2` as in the example code give the splitter enough flexibility to *copy* the data to S rather than locating the fields there. Whether S or T is the right model for the trusted host depends on the scenario; what is important is that the security policy is automatically verified in each case.

4.3 Declassification Constraints

Consider the oblivious transfer example from Alice's point of view. She has two private pieces of data, and she is willing to release exactly one of the two to Bob. Her decision to declassify the data is dependent on Bob not having requested the data previously. In the example program, this policy is made explicit in two ways. First, the method `transfer` explicitly declares that it uses her authority, which is needed to perform the declassification. Second, the program itself tests (in line 10) whether `transfer` has been invoked previously—presumably Alice would not have given her authority to this program without this check to enforce her policy.

This example shows that it is not enough simply to require that any `declassify` performed on Alice's behalf executes on a host she trusts to hold the data. Alice also must be confident that the decision to perform the declassification, that is, the program execution leading to the `declassify`, is performed correctly.

The program counter label summarizes the information dependencies of the decision to arrive at the corresponding program point. Thus, a `declassify` operation using the authority of a set of principals P introduces the integrity constraint: $I(\underline{\mathtt{pc}}) \sqsubseteq \mathtt{I}_P$ where $\mathtt{I}_P$ is the label $\{\mathtt{?}\!:\!\mathtt{p}_1, \ldots, \mathtt{p}_n\}$ for $\mathtt{p}_i \in P$. This constraint says that each principal `p` whose authority is needed to perform the declassification must trust that the program has reached the `declassify` correctly.

Returning to the oblivious transfer example, we can now explain the need to use the `endorse` operation. Alice's authority is needed for the declassification, but, as described above, she must also be sure of the integrity of the program counter when the program does the declassification. Omitting the `endorse` when testing `n` on line 12 would lower the integrity of the program counter within the branches—Alice doesn't trust that `n` was computed correctly, as indicated by its (lack of an) integrity label on line 6. She must add her endorsement to `n`, making explicit her agreement with Bob that she doesn't need to know `n` to enforce her security policy.

Using the static constraints just described, the splitter finds a set of possible hosts for each field and statement. This process may yield many solutions, or none at all—for instance, if the program manipulates data too confidential for any known host. When no solution exists, the splitter gives an error indicating which constraint is not satisfiable. We have found that the static program analysis is remarkably useful in identifying problems with apparently secure programs. When more than one solution exists, the splitter chooses hosts to optimize performance of the distributed system, as described in Section 6.

5 Dynamic Enforcement

In the possible presence of bad hosts that can fabricate messages, run-time checks are required to ensure security. For example, access to an object field on a remote host must be authenticated to prevent illegal data transfers from occurring. Thus, the information-flow policy is enforced by a combination of static constraints (controlling how the program is split) and dynamic checks to ensure that running program obeys the static constraints.

When a program is partitioned, the resulting partitions contain both ordinary code to perform local computation and calls to a special run-time interface that supports host communication. Figure 3 shows the interface to the distributed run-time system.[3] There are three operations for transferring data between hosts: `getField`, `setField`, and `forward`; and three operations for transferring control between hosts: `rgoto`, `lgoto`, and `sync`. These operations define building blocks for a protocol that exchanges information among the hosts running partitions.

[3]We have simplified this interface for clarity; for instance, the actual implementation provides direct support for array manipulation.

```
Val  getField(HostID h, Obj o, FieldID f)
Val  setField(HostID h, Obj o, FieldID f, Val v)
void forward(HostID h, FrameID f, VarID var, Val v)
void rgoto(HostID h, FrameID f, EntryPt e, Token t)
void lgoto(Token t)
Token sync(HostID h, FrameID f, EntryPt e, Token t)
```

Figure 3: Run-time interface

The `rgoto` and `lgoto` control operations are primitive constructs for transferring control from one program point to another that is located on a different host. In general a program partition comprises a set of code fragments that offer entry points to which `rgoto` and `lgoto` transfer control. These two kinds of goto operations are taken from a low-level security-typed language for which it has been proven that every well-typed program automatically enforces noninterference [50].

The run-time interface describes all the ways that hosts can interact. To show that bad hosts cannot violate the security assurance provided by the system, it is therefore necessary to consider each of the run-time operations in turn and determine what checks are needed to enforce the assurance condition given in Section 3.2.

5.1 Access Control

The simplest operations provided by the run-time interface are `getField` and `setField`, which perform remote field reads and writes. Both operations take a handle to the remote host, the object that contains the field, and an identifier for the field itself. The `setField` operation also takes the value to be written.

These requests are dispatched by the run-time system to the appropriate host. Suppose h_1 sends a field access request to h_2. Host h_2 must perform an access control check to determine whether to satisfy the request or simply ignore it, while perhaps logging any improper request for auditing purposes. A read request for a field f labeled $\mathtt{L}_f$ is legal only if $C(\mathtt{L}_f) \sqsubseteq \mathtt{C}_{h_1}$, which says that h_1 is trusted enough to hold the data stored in f. Similarly, when h_1 tries to update a field labeled $\mathtt{L}_f$, h_2 checks the integrity constraint $\mathtt{I}_{h_1} \sqsubseteq I(\mathtt{L}_f)$, which says that the principals who trust f also trust h_1. These requirements are the dynamic counterpart to those used for host selection (see Section 4.1).

Note that because field and host labels are known at compile time, an access control list can be generated for each field, and thus label comparisons can be optimized into a single lookup per request. There is no need to manipulate labels at run time.

5.2 Data Forwarding

Another difficulty with moving to a distributed setting is that the run-time system must provide a mechanism to pass data between hosts without violating any of the confidentiality policies attached to the data. The problem is most easily seen when there are three hosts and the control flow $h_1 \longrightarrow l \longrightarrow h_2$: execution starts on h_1, transfers to l, and then completes on h_2. Hosts h_1 and h_2 must access confidential data d (and are trusted to do so), whereas l is not allowed to see d. The question is how to make d securely available to h_2. Clearly it is not secure to transfer d in plaintext between the trusted hosts via l.

There are essentially two solutions to this problem: pass d via l in encrypted form, or forward d directly to h_2. We chose to implement the second solution. After hosts have been assigned, the splitter infers statically where the data forwarding should occur, using a standard definition-use dataflow analysis. The run-time interface provides an operation `forward` that permits a local variable to be forwarded to a particular stack frame on a remote host. The same mechanism is used to transmit a return value to a remote host. Data forwarding requires that the recipient validate the sender's integrity, as with `setField`.

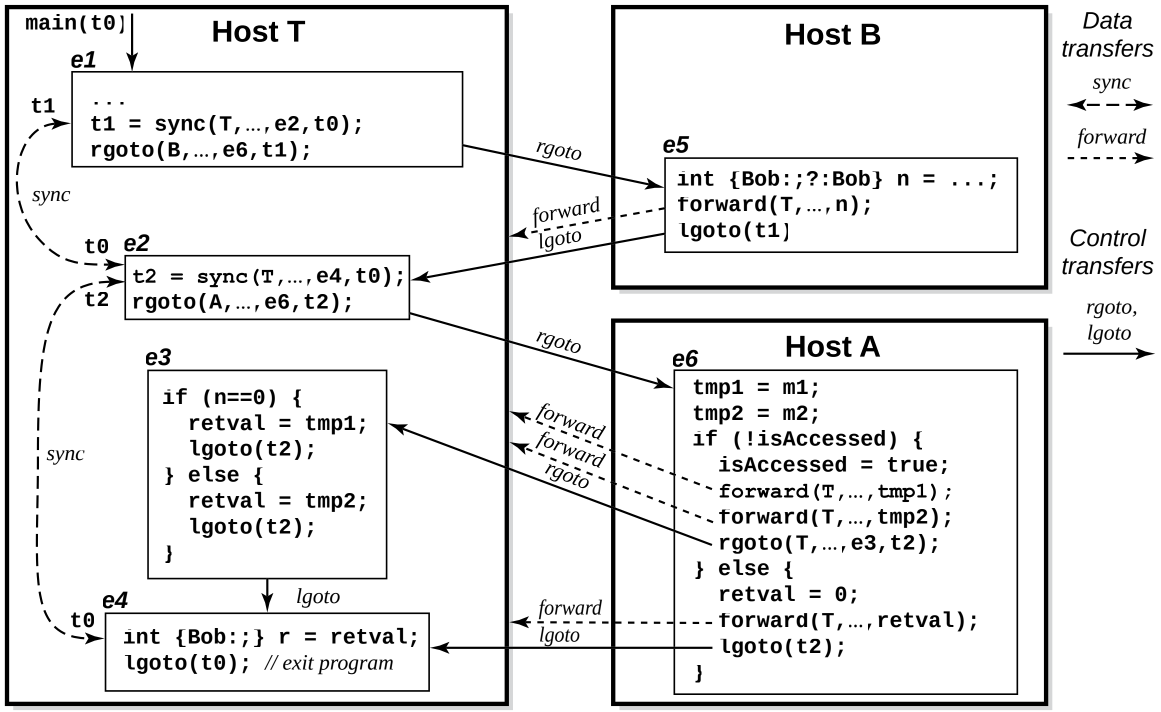

Figure 4: Control flow graph of the oblivious transfer program

5.3 Control Transfer Integrity

So far, we have not addressed the issue of concurrency, which is inherently a concern for security in distributed systems. The problem of protecting confidentiality in a concurrent setting is difficult [47, 42], and we do not attempt to solve the general case here. Instead, we take advantage of the single-threaded nature of the source program by using the idea that the integrity of the program counter obeys a stack discipline.

Consider a scenario with three hosts: h_1 and h_2 have high integrity, and l has relatively lower integrity (that is, its integrity is not equal to or greater than that of h_1 or h_2). Because the program has been partitioned into code fragments, each host is prepared to accept control transfers at multiple entry points, each of which begins a different code fragment. Some of the code fragments on h_1 and h_2 make use of the greater privilege available due to higher integrity (e.g., the ability to declassify certain data).

Suppose the source program control flow indicates control transfer in the sequence $h_1 \longrightarrow l \longrightarrow h_2$. A potential attack is for l to improperly invoke a privileged code fragment residing on h_2, therefore violating the behavior of the original program and possibly corrupting or leaking some data. Hosts h_1 and h_2 can prevent these attacks by simply denying l the right to invoke entry points that correspond to privileged code, but this strategy prevents h_2 from using its higher privileges after control has passed through l—even if this control transfer was supposed to occur according to the source program.

We have developed a mechanism to prevent these illegal control transfers, based on a stack discipline for manipulating capabilities. The intuition is that the block structure and sequential behavior of the source program, which are embodied at run-time by the stack of activation records, induce a similar LIFO property on the program counter's integrity. The deeper the stack, the more data the program counter depends on, and consequently, the lower its integrity.

This correspondence between stack frames and $\underline{pc}$ integrity is not perfect because the $\underline{pc}$ label need not decrease in lock step with every stack frame. A single stack frame may be used by a block of code that is partitioned across several hosts of differing integrity, for example. Nevertheless, this correspondence suggests that we use a stack discipline based on integrity to regulate control transfers. To distinguish between the stack of activation records (whose elements are represented by `FrameID` objects) and the stack of host control transfers, we refer to the latter as the ICS—integrity control stack.

Informally, in the scenario above, the first control transfer (from h_1 to l) pushes a capability for return to h_2 onto the ICS, after which computation is more restricted (and hence may reside on a less trusted machine). The second control transfer (from l to h_2) consumes the capability and pops it off the ICS, allowing h_2 to regain its full privileges. The idea is that before transferring control to l, trusted machines h_1 and h_2 agree that the only valid, privileged entry point between them is the one on h_2. Together, they generate a capability for the entry point that h_1 passes to l on the first control transfer. Host l must present this capability before being granted access to the more privileged code. Illegal attempts to transfer control from l to h_1 or to h_2 are rejected because h_1 and h_2 can validate the (unique) capability to transfer control from l.

5.4 Example Control Flow Graph

Figure 3 shows the signatures for the three control transfer facilities: `rgoto` (for "regular" control transfers that do not affect the ICS), `lgoto` (for "LIFO" transfers—ICS pops), and `sync` (for generating capabilities—ICS pushes). The capabilities are represented as `Token` objects. In addition to the code fragment to be jumped to (given by the `EntryPt` argument), control transfer is to a specific stack frame (given by `FrameID`) on a particular host.

We describe in detail the operation of these mechanisms in the next section, but first it is helpful to see an example of their use.

Figure 4 shows the control-flow graph of a possible splitting of the oblivious transfer example in a host environment that contains Alice's machine A, Bob's machine B and the partially trusted server, T from Section 3.1. We have chosen this simple example because it presents an interesting partitioning without being too large to describe here. For completeness, we describe the unoptimized behavior; optimizations that affect the partitioning process and run-time performance are discussed in Sections 6 and 7.

For lack of space, we show only a fragment of the `main`[4] method. Host T initially has control and possesses a single capability `t0`, which is on top of the ICS. Bob's host is needed to initialize `n`—his choice of Alice's two fields. Recall that $\{$`?:Bob`$\} \not\sqsubseteq \{$`?:Alice`$\}$, which means that B is relatively less trusted than T. Before transferring control to B, T `sync`'s to a suitable return point (entry `e2`), which pushes a new capability, `t1`, onto the ICS (hiding `t0`). The `sync` operation then returns this fresh capability token, `t1`, to `e1`.

Next, T passes `t1` to entry point `e5` on B via `rgoto`. There, Bob's host computes the value of `n` and returns control to T via `lgoto`, which requires the capability `t1` to return to a host with relatively higher integrity. Upon receiving this valid capability, T pops `t1`, restoring `t0` as the top of the ICS. If instead B maliciously attempts to invoke any entry point on either T or A via `rgoto`, the access control checks deny the operation. The only valid way to transfer control back to T is by invoking `lgoto` with one-time capability `t1`. Note that this prevents Bob from initiating a race to the assignment on line `11` of the example, which might allow two of his transfer requests (one for `m1` and one for `m2`) to be granted and thus violate Alice's declassification policy.

Alice's machine must check the `isAccessed` field, so after B returns control, T next `sync`s with the return point of `transfer` (the entry point `e4`), which again pushes new token `t2` onto the ICS. T then transfers control to `e6` on A, passing `t2`. The entry point `e6` corresponds to the beginning of the `transfer` method.

Alice's machine performs the comparison, and either denies access to Bob by returning to `e4` with `lgoto` using `t2`, or forwards the values of `m1` and `m2` to T and hands back control via `rgoto` to `e3`, passing the token `t2`. If Bob has not already made a request, T is able to check `n` and assign the appropriate value of `tmp1` and `tmp2` to `retval`, then jump to `e4` via `t2`. The final block shows T exiting the program by invoking the capability `t0`.

[4]We omitted the `main` method and constructors from Figure 2 to simplify the presentation; they contain simple initialization code. We also omit the details of `FrameID` objects, which are unimportant for this example.

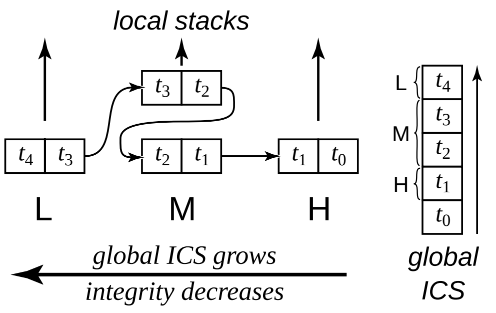

Figure 5: Distributed implementation of the global stack

5.5 Control Transfer Mechanisms

This section describes how rgoto, lgoto, and sync manipulate the ICS, which is itself distributed among the hosts, and defines the dynamic checks that must occur to maintain the desired integrity invariant.

A capability token t is a tuple $\{h, f, e\}_{k_h}$ containing a HostID, a FrameID, and an EntryPt. To prevent forgery and ensure uniqueness, the tuple is appended to its hash with h's private key and a nonce.

The global ICS is represented by a collection of local stacks, as shown in Figure 5. Host h's local stack, s_h, contains pairs of tokens $(\mathtt{t}, \mathtt{t}')$ as shown. The intended invariant is that when the top of h's stack, $\mathtt{top}(s_h)$, is $(\mathtt{t}, \mathtt{t}')$, then t is the token most recently issued by h. Furthermore, the only valid lgoto request that h will serve must present the capability t. The other token, t′, represents the capability for the next item on the global stack; it is effectively a pointer to the tail of the global ICS.

To show that these distributed stacks enforce a global stack ordering on the capabilities, we prove a stronger invariant of the protocol operations (see Appendix A). Whenever control is transferred to low-integrity hosts, there is a unique re-entry point on high-security hosts that permits high-integrity computation. This uniqueness ensures that if a low-integrity host is bad, it can only jeopardize the security of low-integrity computation.

The recipients of control transfer requests enforce the ordering protocol. Assume the recipient is the host h, and the initiator of the request is i. The table in Figure 6 specifies h's action for each type of request. We write $e(f, \mathtt{t})$ for local invocation of the code identified by entry point e in stack frame f, passing the token t as an additional argument.

This approach forces a stack discipline on the integrity of the control flow: rgoto may be used to transfer control to an entry point that requires lesser or equal integrity; lgoto may transfer control to a higher-integrity entry point—provided that the higher-integrity host previously published a capability to that entry point. These capabilities can be used at most once: upon receiving an lgoto request using the valid capability t, h pops its local capability stack, thereby invalidating t for future uses. Calls to sync and lgoto thus come in pairs, with each lgoto consuming the capability produced by the corresponding sync.

Just as we have to dynamically prevent malicious hosts from improperly accessing remote fields, we must also ensure that bad hosts cannot improperly invoke remote code. Otherwise, malicious hosts could indirectly violate the integrity of data affected by the code. Each entry point e has an associated dynamic access control label $\mathtt{I}_e$ that regulates the integrity of machines that may remotely invoke e. The receiver of an rgoto or sync request checks the integrity of the requesting host against $\mathtt{I}_e$ as shown in Figure 6. The label $\mathtt{I}_e$ is given by $(\bigsqcap_{v \in D(e)} \mathtt{L}_v) \sqcap \mathtt{I}_P$, where $D(e)$ is the set of variables and fields written to by the code in e and $\mathtt{I}_P$ is the integrity label of the principals, P, whose authority is needed to perform any declassifications in e.

The translation phase described in the next section inserts control transfers into the source program. To prevent confidentiality and integrity policies from being violated by the communications of the transfer mechanisms themselves, there are constraints on where rgoto and sync may be added.

Suppose a source program entry point e is assigned to host i, but doing so requires inserting an rgoto or sync

rgoto$(h, f, e, \mathtt{t})$	Transfers control to the entry point e in frame f on the host h. Host i's current capability t is passed to h.	if $(\mathtt{I}_i \sqsubseteq \mathtt{I}_e)$ {
		  $e(f, \mathtt{t})$;
		} else ignore;

lgoto(t)
 	
(where $\mathtt{t} = \{h, f, e\}_{k_h}$)
	Pops h's local control stack after verifying the capability t; control moves to entry point e in frame f on host h, restoring privileges.	if (top(s_h)==(t,t$'$)) {
		  pop(s_h);
		  $e(f, \mathtt{t}')$;
		} else ignore;

sync$(h, f, e, \mathtt{t})$	Host h checks i's integrity; if sufficient, h returns to i a new capability (nt) for entry point e in frame f.	if $(\mathtt{I}_i \sqsubseteq \mathtt{I}_e)$ {
		  nt = $\{h, f, e\}_{k_h}$;
		  push$(s_h$, (nt, t));
		  send$(i$, nt);
		} else ignore;

Figure 6: Host h's response to transfer requests from host i

to another entry point e' on host h. The necessary constraints are:

$$C(\underline{\mathtt{pc}}) \sqsubseteq \mathtt{C}_h \qquad \mathtt{I}_i \sqsubseteq \mathtt{I}_{e'} \qquad \mathtt{I}_e \sqsubseteq \mathtt{I}_{e'}.$$

The first inequality says that i can't leak information to h by performing this operation. The second inequality says that host i has enough integrity to request this control transfer. This constraint implies that the dynamic integrity checks performed by h are guaranteed to succeed for this legal transfer—the dynamic checks are there to catch malicious machines, not well-behaved ones. Finally, the third constraint says that the code of the entry point e itself has enough integrity to transfer the control to e'. Furthermore, because sync passes a capability to h, it requires the additional constraint that $I_h \sqsubseteq I(\underline{\mathtt{pc}})$, which limits the damage h can do by invoking the capability too early, thus bypassing the intervening computation.

These enforcement mechanisms do not attempt to prevent denial of service attacks, as such attacks do not affect confidentiality or integrity. These measures *are* sufficient to prevent a bad low-integrity host from launching race-condition attacks against the higher integrity ones: hosts process requests sequentially, and each capability offers one-shot access to the higher integrity hosts.

While our restrictive stack-based control transfer mechanism is *sufficient* to provide the security property of Section 3.2, it is not *necessary*; there exist secure systems that lie outside the behaviors expressible by the ICS. However, following the stack discipline is sufficient to express many interesting protocols that move the thread of control from trusted hosts to untrusted hosts and back. Moreover, the splitter determines when a source program can obey the stack ordering and generates the protocol automatically.

6 Translation

Given a program and host configuration, the splitting translation is responsible for assigning a host to each field and statement. The Jif/split compiler takes as input the annotated source program and a description of the known hosts. It produces as output a set of Java files that yield the final split program when compiled against the run-time interface. There are several steps to this process.

In addition to the usual typechecking performed by an ordinary Java compiler, the Jif/split front end collects security label information from the annotations in the program, performing label inference when annotations are omitted. This process results in a set of label constraints that capture the information flows within the program. Next, the compiler computes a set of possible hosts for each statement and field, subject to the security constraints described in Section 4. If no host can be found for a field or statement, the splitter conservatively rejects the program as being insecure.

There may also be many valid host assignments for each field or statement, in which case performance drives the host selection process. The splitter uses dynamic programming to synthesize a good solution by attempting to minimize the number of remote control transfers and field accesses, two operations that dominate run-time overhead. The algorithm works on a weighted control-flow graph of the program; the weight on an edge represents an approximation to the run-time cost of traversing that edge.

This approach also has the advantage that principals may indicate a preference for their data to stay on one of severally equally trusted machines (perhaps for performance reasons) by specifying a lower cost for the preferred machine. For example, to obtain the example partition shown in Figure 4, Alice also specifies a preference for her data to reside on host A, causing fields `m1`, `m2`, and `isAccessed` to be located on host A. Without the preference declaration, the optimizer determines that fewer network communications are needed if these fields are located at T instead. This alternative assignment is secure because Alice trusts the server equally to her own machine.

After host selection, the splitter inserts the proper calls to the runtime, subject to the constraints described in Section 5. An `lgoto` must be inserted exactly once on every control flow path out of the corresponding `sync`, and the `sync`–`lgoto` pairs must be well nested to guarantee the stack discipline of the resulting communication protocol. The splitter also uses standard dataflow analysis techniques to infer where to introduce the appropriate data forwarding.

Finally, the splitter produces Java files that contain the final program fragments. Each source Jif class `C` translates to a set of classes `C$Host`$_i$, one for each known host $h_i \in H$. In addition to the translated code fragments, each such class contains the information used by the runtime system for remote references to other classes. The translation of a field includes accessor methods that, in addition to the usual get and set operations, also perform access control checks (which are statically known, as discussed in Section 4). In addition, each source method is represented by one frame class per host. These frame classes correspond to the `FrameID` arguments needed by the runtime system of Figure 3; they encapsulate the part of the source method's activation record visible to a host.

7 Implementation

We have implemented the splitter and the necessary run-time support for executing partitioned programs. Jif/split was written in Java as a 7400-line extension to the existing Jif compiler. The run-time support library is a 1700-line Java program. Communication between hosts is encrypted using SSL (the Java Secure Socket Extension (JSSE) library, version 1.0.2) [18]. To prevent forging, tokens for entry points are hashed using the MD5 implementation from the Cryptix library, version 3.2.0 [5].

To evaluate the impact of our design, we implemented several small, distributed programs using the splitter. Because we are using a new programming methodology that enforces relatively strong security policies, direct comparison with the performance of other distributed systems was difficult; our primary concern was security, not performance. Nevertheless, the results are encouraging.

7.1 Benchmarks

We have implemented a number of programs in this system. The following four are split across two or more hosts:

- **List** compares two identical 100 element linked lists that must be located on different hosts because of confidentiality. A third host traverses the lists.
- **OT** is the oblivious transfer program described earlier in the paper. One hundred transfers are performed.
- **Tax** simulates a tax preparation service. A client's trading records are stored on a stockbroker's machine. The client's bank account is stored at a bank's machine. Taxes are computed by a tax preparer on a third host. The principals have distinct confidentiality concerns, and `declassify` is used twice.

Metric	List	OT	Tax	Work	OT-h	Tax-h
Lines	110	50	285	45	175	400
Elapsed time (sec)	0.51	0.33	0.58	0.49	0.28	0.27
Total messages	1608	1002	1200	600	800	800
`forward` ($\times 2$)	400	101	300	0	-	-
`getField` ($\times 2$)	2	100	0	0	-	-
`lgoto`	402	200	0	300	-	-
`rgoto`	402	400	600	300	-	-
Eliminated ($\times 2$)	402	600	400	300	-	-

Table 1: Benchmark measurements

- **Work** is a compute-intensive program that uses two hosts but communicates relatively little.

Writing these programs requires adding security policies (labels) to some type declarations from the equivalent single-machine Java program. These annotations are 11–25% of the source text, which is not surprising because the programs contain complex security interactions and little real computation.

7.2 Experimental Setup

Each subprogram of the split program was assigned to a different physical machine. Experiments were run on a set of three 1.4 GHz Pentium 4 PCs with 1GB RAM running Windows 2000. Each machine is connected to a 100 Mbit/second Ethernet by a 3Com 3C920 controller. Round-trip ping times between the machines average about 310 μs. This LAN setting offers a worst-case scenario for our analysis—the overheads introduced by our security measures are relatively more costly than in an Internet setting. Even for our local network, network communication dominates performance. All benchmark programs were run using SSL, which added more overhead: the median application-to-application round-trip time was at least 640 μs for a null Java RMI [37] call over SSL.

All benchmarks were compiled with version 1.3.0 of the Sun `javac` compiler, and run with version 1.3.0 of the Java HotSpot Client VM. Compilation and dynamic-linking overhead is not included in the times reported.

7.3 Results

For all four benchmarks, we measured both running times and total message counts so that performance may be estimated for other network configurations. The first row of Table 1 gives the length of each program in lines of code. The second row gives the median elapsed wall-clock time for each program over 100 trial runs. The following rows give total message counts and a breakdown of counts by type (`forward` and `getField` calls require two messages). The last row shows the number of `forward` messages eliminated by piggybacking optimizations described below.

For performance evaluation, we used Java RMI to write reference implementations of the Tax and OT programs and then compared them with our automatically generated programs. These results are shown in the columns OT-h and Tax-h of Table 1. Writing the reference implementation securely and efficiently required some insight that we obtained from examining the corresponding partitioned code. For example, in the OT example running on the usual three-host configuration, the code that executes on Alice's machine should be placed in a critical section to prevent Bob from using a race condition to steal both hidden values. The partitioned code automatically prevents the race condition.

The hand-coded implementation of OT ran in 0.28 seconds; the automatically partitioned program ran in 0.33 seconds, a slowdown of 1.17. The hand-coded version of Tax also ran in 0.27 seconds; the partitioned program ran in 0.58 seconds, a slowdown of 2.17. The greater number of messages sent by the partitioned programs explains most of this slowdown. Other sources of added overhead turn out to be small:

- Inefficient translation of local code
- Run-time checks for incoming requests
- MD5 hashing to prevent forging and replaying of tokens

The prototype Jif/split compiler attempts only simple optimizations for the code generated for local use by a single host. The resulting Java programs are likely to have convoluted control flow that arises as an artifact of our translation algorithm—the intermediate representation of the splitter resembles low-level assembly code more than Java. This mismatch introduces overheads that the hand-coded programs do not incur. The overhead could be avoided if Jif/split generated Java bytecode output directly; however, we leave this to future work.

Run-time costs also arise from checking incoming requests and securely hashing tokens. These costs are relatively small: The cost of checking incoming messages is less than 6% of execution time for all four example programs. The cost of token hashing accounted for approximately 15% of execution time across the four benchmarks. Both of these numbers scale with the number of messages in the system. For programs with more substantial local computations, we would expect these overheads to be less significant.

For a WAN environment, the useful point of comparison between the hand-coded and partitioned programs is the total number of messages sent between hosts. Interestingly, the partitioned Tax and OT programs need fewer messages for control transfers than the hand-coded versions. The hand-coded versions of OT and Tax each require 400 RMI invocations. Because RMI calls use two messages, one for invocation and one for return, these programs send 800 messages. While the total messages needed for the Jif/split versions of OT and Tax are 1002 and 1200, respectively, only 600 of these messages in each case are related to control transfers; the rest are data forwards. The improvement over RMI is possible because the `rgoto` and `lgoto` operations provide more expressive control flow than procedure calls. In particular, an RMI call must return to the calling host, even if the caller immediately makes another remote invocation to a third host. By contrast, an `rgoto` or `lgoto` may jump directly to the third host. Thus, in a WAN environment, the partitioned programs are likely to execute more quickly than the hand-coded program because control transfers should account for most of the execution time.

7.4 Optimizations

Several simple optimizations improve system performance:

- Calls to the same host do not go through the network.
- Hashes are not computed for tokens used locally to a host.
- Multiple data forwards to the same recipient are combined into a single message and also piggybacked on `lgoto` and `rgoto` calls when possible. As seen in Table 1, this reduces `forward` messages by more than 50% (the last row is the number of round trips eliminated).

A number of further simple optimizations are likely to be effective. For example, much of the performance difference between the reference implementation of OT and the partitioned implementation arises from the server's ability to fetch the two fields `m1` and `m2` in a single request. This optimization (combining `getField` requests) could be performed automatically by the splitter as well.

Currently, `forward` operations that aren't piggybacked with control transfers require an acknowledgment to ensure that all data is forwarded before control reaches a remote host. It is possible to eliminate the race condition that necessitates this synchronous data forwarding. Because the splitter knows statically what forwards are expected at every entry point, the generated code can block until all forwarded data has been received. Data transfers that are not piggybacked can then be done in parallel with control transfers. However, this optimization has not been implemented.

8 Trusted Computing Base

An important question for any purported security technique is the size and complexity of the *trusted computing base* (TCB). All else being equal, a distributed execution platform suffers from a larger TCB than a corresponding single-host execution platform because it incorporates more hardware and software. On the other hand, the architecture described here may increase the participants' confidence that trustworthy hosts are being used to protect their confidentiality.

What does a principal p who participates in a collaborative program using this system have to trust? The declaration signed by p indicates to what degree p trusts the various hosts. By including a declaration of trust for a host h in the declaration, p must trust the hardware of h itself, the h's operating system, and the splitter run-time support, which (in the prototype implementation) implicitly includes Java's.

Currently, the Jif/split compiler is also trusted. Ongoing research based on certified compilation [26] or proof-carrying code [30] might be used to remove the compiler from the TCB and instead allow the bytecode itself to be verified [20].

Another obvious question about the trusted computing base is to what degree the partitioning process itself must be trusted. It is clearly important that the subprograms a program is split into are generated under the same assumptions regarding the trust relationships among principals and hosts. Otherwise, the security of principal p might be violated by sending code from different partitionings to hosts trusted by p. A simple way to avoid this problem is to compute a one-way hash of all the splitter's inputs—trust declarations and program text—and to embed this hash value into all messages exchanged by subprograms. During execution, incoming messages are checked to ensure that they come from the same version of the program.

A related issue is where to partition the program. It is necessary that the host that generates the program partition that executes on host h be trusted to protect all data that h protects during execution. That is, the partitioning host could be permitted to serve in place of h during execution. A natural choice is thus h itself: each participating host can independently partition the program, generating its own subprogram to execute. That the hosts have partitioned the same program under the same assumptions can be validated using the hashing scheme described in the previous paragraph. Thus, the partitioning process itself can be decentralized yet secure.

9 Related Work

There are two primary areas of research related to this work: static and dynamic enforcement of information-flow policies and support for transparently distributed computation.

There has been much research on end-to-end security policies and mandatory access control in multilevel secure systems. Most practical systems have opted for dynamic enforcement using a mix of mandatory and discretionary access control, for example as described in the Orange Book [9]. These techniques (e.g., [13, 23]) have difficulty controlling implicit information flows accurately.

Static analysis of information flow has a long history, although it has not been as widely used as dynamic checking. Denning originally proposed a language to permit static checking [8], but it was not implemented. Other researchers [24, 25, 12] developed techniques for information-flow checking using formal specifications and automatic or semi-automatic theorem proving.

Recently, there has been more interest in provably-secure programming languages. Palsberg and Ørbæk have developed a simple type system for checking integrity [33]. Others have taken a similar approach to static analysis of secrecy, encoding rules similar to Denning's in a type system and showing them to be sound using programming language techniques [46, 17, 35]. No language of the complexity of Jif [27] has been proven to enforce noninterference; also, extended notions of soundness that encompass declassification are not yet fully developed. All of these previous language-based techniques assume execution on a trusted platform.

Program slicing techniques [44] provide information about the data dependencies in a piece of software. The use of backward slices to investigate integrity and related security properties has been proposed [14, 21], but the focus has been on debugging and understanding existing software.

A number of systems (such as Amoeba and Sprite [10]) automatically redistribute computation across a distributed system to improve performance, though not security. Various transparently distributed programming languages have been developed as well; a good early example is Emerald [19]. Modern distributed interface languages such as CORBA [31] or Java RMI do not enforce end-to-end security policies.

In our approach, certain parts of the system security policy are explicit in the labels appearing in the program; others are implicit in the declassifications and endorsements made in the program text. There has been some work on specifying end-to-end security for systems containing downgrading, such as the work on intransitive noninterference [38, 34] and on robust declassification [49].

Jif and secure program partitioning are complementary to current initiatives for privacy protection on the Internet. For example, the recent Platform for Privacy Preferences (P3P) [32] provides a uniform system for specifying users' confidentiality policies. Security-typed languages such as Jif could be used for the implementation of a P3P-compliant web site, providing the enforcement mechanisms for the P3P policy.

10 Future Work

The Jif/split prototype has given us insight into the difficulties of building distributed systems with strong end-to-end information-flow guarantees, but there is still much room for improvement.

Experience with larger and more realistic programs will be necessary to determine the real trade-offs involved. This paper has focused on one axis of security, namely protecting confidential data. Other axes, such as reliability and auditing of transactions, also play a role in the security of distributed computations, and they should not be neglected.

Of course security and performance are often at odds, and the same is true here. Jif/split assumes that the security of the data is more important than the performance of the system. However, we believe that encoding the security policy in the programming language makes this trade-off more explicit: if the performance of a program under a certain security policy is unsatisfactory, it is possible to relax the policy (for instance, by declaring more trust in certain hosts, or by reducing the restrictions imposed by the label annotations). Under a relaxed policy, the compiler may be able to find a solution with acceptable performance—the relaxed security policy spells out what security has been lost for performance. The prototype allows some control over performance by allowing the user to specify relative costs of communication between hosts. The host assignment tries to find a minimum cost solution, but other constraints could be added—for example, the ability to specify a particular host for a given field.

Another limitation to the current prototype is that it accepts only sequential source programs. Providing information-flow guarantees in concurrent systems is a difficult problem, but one that is important for providing realistic, secure systems. The main obstacle is soundly accounting for information flows that arise due to synchronization of the processes—without imposing restrictions that prohibit useful programs. Another difficulty in the concurrent setting, which we have not addressed in the present work, is the problem of garbage collection.

More immediately, there are a number of useful features of Jif that are not yet supported in Jif/split. Full Jif includes an `actsfor` relation, which allows the program to determine whether one principal has delegated privileges to another, a `switch label` construct and dynamic labels, which allows labels to be compared and manipulated at run time, and label polymorphism, which allows classes to be parameterized by a security level and enables code re-use. Jif also provides support for tracking information flows through exceptions and other non-local control transfers.

Some of these features can be straightforwardly incorporated into Jif/split. The control-transfer mechanisms described in Section 5 are already sufficient to express exceptions and non-local control transfers. Likewise, the `actsfor` construct presents no technical difficulties, and could readily be included. Label polymorphism could be implemented (at the expense of code bloat) by duplicating the code for each instantiation of a parameterized class; we are investigating cleaner solutions. Dynamic labels appear to be the most difficult feature of Jif to provide in Jif/split. The difficulty is that our code-partitioning scheme relies on the label information to transform the program, but dynamic labels aren't known until run time. This problem we leave to future work.

11 Conclusion

This paper presents a language-based technique for protection of confidential data in a distributed computing environment with heterogeneously trusted hosts. Security policy annotations specified in the source program allow the splitter to partition the code across the network by extracting a suitable communication protocol. The resulting distributed system satisfies the confidentiality policies of principals involved without violating their trust in available hosts. The system also enforces integrity policies, which is needed because of the interaction between integrity and confidentiality in the presence of declassification. The Jif/split prototype demonstrates the feasibility of this architecture. Our experience with example programs has shown the benefits of expressing security policies explicitly in the programming language, particularly with respect to catching subtle bugs.

Collaborative computations carried out among users, businesses, and networked information systems continue to increase in complexity, yet there are currently no satisfactory methods for determining whether the end-to-end behavior of these computations respect the security needs of the participants. The work described in this paper is a novel approach that is a useful step towards solving this essential security problem.

Acknowledgments

We would like to thank those who provided useful feedback on earlier drafts of this paper: Kavita Bala, Sunny Gleason, Dan Grossman, Fred Schneider, Gün Sirer, Stephanie Weirich, and the anonymous referees. Many thanks also to the JLtools development team: Matt Harren, Aleksey Kliger, Navin Sastry, and Dan Spoonhower.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[2] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[3] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[5] Cryptix. `http://www.cryptix.org/products/cryptix31/`.

[6] Ivan Damgård, Joe Kilian, and Louis Salvail. On the (im)possibility of basing oblivious transfer and bit commitment on weakened security assumptions. In Jacques Stern, editor, *Advances in Cryptology – Proceedings of EUROCRYPT 99*, LNCS 1592, pages 56–73. Springer, 1999.

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

[8] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[9] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[10] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum. A comparison of two distributed systems: Amoeba and Sprite. *ACM Transactions on Computer Systems*, 4(4), Fall 1991.

[11] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. In R.L. Rivest, A. Sherman, and D. Chaum, editors, *Advances in Cryptology: Proc. of CRYPTO 82*, pages 205–210. Plenum Press, 1983.

[12] Richard J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, SRI International Computer Science Lab, Menlo Park, California, January 1980.

[13] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[14] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 154–163, 1994.

[15] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86, April 1984.

[16] J. W. Gray III and P. F. Syverson. A logical approach to multilevel security of probabilistic systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–176, 1992.

[17] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[18] Java secure socket extension (JSSE). `http://java.sun.com/products/jsse/`.

[19] E. Jul et al. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.

[21] J. R. Lyle, D. R. Wallace, J. R. Graham, K. B. Gallagher, J. P. Poole, and D. W. Binkley. Unravel: A CASE tool to assist evaluation of high integrity software. IR 5691, NIST, 1995.

[22] Heiko Mantel and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. of the 14th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, June 2001.

[23] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[24] Jonathan K. Millen. Security kernel validation in practice. *Comm. of the ACM*, 19(5):243–250, May 1976.

[25] Jonathan K. Millen. Information flow analysis of formal specifications. In *Proc. IEEE Symposium on Security and Privacy*, pages 3–8, April 1981.

[26] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[27] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[28] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[29] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[30] George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.

[31] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[32] Platform for privacy preferences (P3P). `http://www.w3.org/p3p`.

[33] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[34] Sylvan Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symposium on Security and Privacy*, 1995.

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

[36] M. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[37] Java remote method interface (RMI). `http://java.sun.com/products/jdk/rmi/`.

[38] John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.

[39] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[40] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[41] Geoffrey Smith. A new type system for secure information flow. In *CSFW14*, pages 115–125. IEEE Computer Society Press, jun 2001.

[42] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, January 1998.

[43] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. Technical report, Project Athena, MIT, Cambridge, MA, March 1988.

[44] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[45] Dennis Volpano. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, Boston, MA, January 2000.

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

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

[48] Tatu Ylonen. SSH – secure login connections over the Internet. In *The Sixth USENIX Security Symposium Proceedings*, pages 37–42, San Jose, California, 1996.

[49] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of the 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001.

[50] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *Proc. of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61, 2001.

[51] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, Banff, Canada, October 2001. To appear.

A Appendix

This appendix proves that the control-transfer protocols generated by Jif/split protect the integrity of the program counter. The purpose of these protocols is to ensure that at any point in time, the set of (relatively) low-integrity hosts has access to at most one capability that grants access to high-integrity (more privileged) code. Thus, untrusted hosts can jeopardize only low-integrity computation—the control behavior of the high-integrity parts of the split code is the same as in the original, single-host source program. The *Stack Integrity Theorem,* described below, proves that the distributed systems generated by Jif/split satisfy this security invariant.

To arrive at this result, we need to first model the behavior of the system at an appropriate level of detail. There are two parts to this model: First, Jif/split statically produces code fragments to distribute among the hosts. These code fragments obey static constraints imposed by the compiler and splitter, but they also have a run-time effect on the behavior of the system—for instance, a code fragment may terminate in a control transfer to a different host. Second, the run-time system of each host manipulates stacks of capability tokens that are used for dynamic checking. The combination of static constraints on the partitions created by the splitter and dynamic checks performed by the run-time system protects the control-transfer integrity.

A.1 Hosts

Let H be a set of known hosts $\{h_1, \ldots, h_n\}$. We assume that each host h has an associated integrity label $\mathtt{I}_h$. Fix an integrity label ι, used to define the relative trust levels between hosts. Let $H_G = \{h \mid \mathtt{I}_h \sqsubseteq \iota\}$ be the set of *good hosts*, and let $H_B = \{h \mid \mathtt{I}_h \not\sqsubseteq \iota\}$ be the set of *bad hosts*. For example, with respect to a single principal, $\mathtt{p}$, we might choose $\iota = \{\mathtt{?:p}\}$. In this case, H_G is the set of hosts trusted by $\mathtt{p}$ and H_B is the set of hosts not trusted by $\mathtt{p}$. Note that $H_G \cup H_B = H$. Throughout this appendix, we call objects with label $\not\sqsubseteq \iota$ *bad* and objects with label $\sqsubseteq \iota$ *good*. [5]

We assume that good hosts follow the protocols and that bad hosts might not. In particular, bad hosts may attempt to duplicate or otherwise misuse the capability tokens; they may also generate spurious messages that contain tokens previously seen by any bad host.

The run-time system provided also ensures that good hosts execute requests atomically. In particular, a host h that is executing the sequential code fragment corresponding to an entry point e_h will not be executing code for any other entry point e'_h on h. This assumption justifies the state-transition approach described below, because we show that the local processing on each host, if performed atomically, preserves a global invariant. Thus, even though several good hosts may be executing concurrently—perhaps because they are responding to low-integrity requests generated by bad hosts—the concurrency does not affect high-integrity computation. The Stack Integrity Theorem establishes that high-integrity computation is still single-threaded, despite the possible concurrency introduced by bad hosts.

[5] Recall that in the integrity lattice, labels representing more integrity are lower in the $\sqsubseteq$ order.

A.2 Modeling Code Partitions

To capture the static constraints on the behavior of good hosts, we define the notion of an *entry point*: an entry point e is a the name of a code partition generated by the splitter—it can be thought of as a remote reference to a single-threaded piece of program that resides entirely on one host. An entry point names a program point to which control may be transferred from a remote host. Each entry point e has an associated integrity label $\mathtt{I}_e$ as described in Section 5. Note that a low-integrity entry point may be located at a high-integrity machine. Let E be the set of entry points generated by a given program. and let E_h be the set of entry points located on host h.

Because our proof is concerned with the control transfers between hosts, we can ignore the details of the sequential code named by an entry point. Consequently, an entry point e on a good host $h \in H_G$ can be thought of as a function that takes a frame f and a token t and produces an *action*, which is a pair (h, a) of the host h and an operation a, in one of the following forms:

$$e(f,t) = \begin{cases} (h, \mathtt{rgoto}(h', f', e', t)) & h \text{ transfers control to entry } e' \text{ on } h' \text{ in frame } f'. \\ (h, \mathtt{sync}(h', f', e', t)) & h \text{ requests a sync with entry } e' \text{ on } h' \text{ and frame } f'; h \text{ blocks until reply.} \\ (h, \mathtt{lgoto}(t)) & h \text{ transfers control to entry } e' \text{ on } h' \text{ in frame } f' \text{ if } t = \{h', f', e'\}_{k_{h'}} \end{cases}$$

Here, t is a capability token, which is a tuple $\{h, f, e\}_{k_h}$ consisting of a host identifier, a frame identifier, and an entry point identifier. To prevent forgery and ensure uniqueness of such tokens, the tuple is appended to its hash with h's private key and a nonce. Thus a token of this form can be generated only by host h, but its validity can be checked by any host with h's public key. (In the text below, we use t, t', u, u', etc., to range over tokens.)

The `sync` operation is the only control transfer mechanism that involves an exchange of messages between hosts. (Both `rgoto` and `lgoto` unconditionally transfer control to the recipient.) Because the initiator (host h in the above description) of a `sync` request expects the recipient (h') to respond with a freshly generated token, h blocks until it gets h''s reply. For the purposes of analysis, we treat h's behavior after issuing a `sync` request to h' as an entry point $\mathit{send}(h')_h$ on h to which h' may return control. The integrity of the entry point $\mathit{send}(h')_h$ is the integrity $\mathtt{I}_e$ of the entry point containing the `sync` operation.

Just like any other entry point, $\mathit{send}(h')_h$ is a function that takes a frame and token and returns an action. In the case that h' is also a good host, and hence follows the appropriate protocol, h' will generate a fresh token t' and return control to h causing the entry point $\mathit{send}(h')_h(f, t')$ to be invoked.

For example, the code fragment below, located on a host h, includes two entry points, e and $\mathit{send}(h')_h$:

```
          e :  x = x+1;
               z = x-2;
               sync(h', f', e', t);
 send(h')_h :  y = y +2;
               rgoto(h'', f'', e'', t');
```

These entry points are modeled as functions $e(f,t) = \mathtt{sync}(h', f', e', t)$ and $\mathit{send}(h')_h(f,t') = \mathtt{rgoto}(h'', f'', e'', t')$, respectively. When e is invoked, the local computations are performed and then host h sends a `sync` request to h', causing h' to block (waiting for the response to be sent by h' to the entry point $\mathit{send}(h')_h$.

Note that this description of entry points is valid only for good hosts—bad hosts may do anything they like with the tokens they receive. For good hosts, the code generated by our splitter satisfies the above abstract specification by construction. For instance, the splitter always terminates the thread of control on a host by inserting a single `rgoto` or an `lgoto`—the splitter never generates several control transfers in a row from the same host (which would cause multiple remote hosts to start executing concurrently). As discussed in Section 5, the splitter also follows some constraints about where `rgoto` and `sync` can be inserted. In order for host h to perform an `rgoto` or `sync` at an entry point e to entry point e' on host h', the following static constraint must be satisfied: $\mathtt{I}_h \sqcup \mathtt{I}_e \sqsubseteq \mathtt{I}_{e'}$. The `sync` operation requires an additional constraint $\mathtt{I}_{h'} \sqsubseteq I(\underline{\mathtt{pc}})$, which limits the damage h' can do by invoking the capability too early, bypassing the intervening computation. Since we assume good hosts are not compromised, `sync`s or `rgoto`s issued by good hosts satisfy these label inequalities.

A.3 Modeling the Run-time Behavior

To capture the dynamic behavior of the hosts, we need to model the state manipulated by the run-time system. Let T be the set of all possible tokens. For the purposes of this proof, we assume that a host may generate a fresh, unforgeable token not yet used anywhere in the system. In practice, this is accomplished by using nonces. The run-time system's local state on a host h includes a *token stack.* which is a list of pairs of tokens

$$(t_1, t_1') : (t_2, t_2') : \ldots : (t_n, t_n').$$

Because only good hosts are trusted to follow the protocol, only good hosts necessarily have token stacks. For each $h \in H_G$ we write s_h for the local token stack associated with the good host h.

We use the notation $\mathit{top}(s_h)$ to denote the top of the stack s_h: the pair (t_n, t_n'). If s_h is empty, then $\mathit{top}(s_h)$ is undefined. For the pair (t_n, t_n'), let $\mathit{fst}(t_n, t_n') = t_n$ be the first projection and $\mathit{snd}(t_n, t_n') = t_n'$ be the second projection. We overload the meaning of *fst* and *snd* to include entire stacks: $\mathit{fst}(s_h) = \{t_1, t_2, \ldots, t_n\}$ and $\mathit{snd}(s_h) = \{t_1', t_2', \ldots, t_n'\}$ when $s_h = \epsilon : (t_1, t_1') : \ldots : (t_n, t_n')$.

When a good host h receives a request from initiator i, h responds to the message according to the following table. (We can't say anything about what a bad host does upon receiving a message, except observe that the bad host may gain access to new tokens.) The *pop* and *push* operations manipulate h's local stack.

$(i, \texttt{rgoto}(h, f, e, t))$	$(i, \texttt{lgoto}(t))$ where $t = \{h, f, e\}_{k_h}$	$(i, \texttt{sync}(h, f, e, t))$

`if (`$I_i \sqsubseteq I_e$`) {`
  $e(f, t)$
`} else ignore`
	`if (`$\mathit{top}(s_h)$ `=` (t, t')`) {`
	  $\mathit{pop}(s_h)$;
	  $e(f, t')$;
	`} else ignore`
		`if (`$I_i \sqsubseteq I_e$`) {`
		  $nt = \{h, f, e\}_{k_h}$;
		  $\mathit{push}(s_h, (nt, t))$;
		  $\mathit{send}(h)_i(f, nt)$;
		`} else ignore`

Note that h will respond to a `lgoto` request only if the token used to generate the request is on top of its local stack. The point of our protocol is to protect against any bad host (or set of bad hosts) causing more than one these "active tokens" to be used at time.

The global state of the system at any point in time is captured by a *configuration*, which is a tuple $\langle s, R, T_R \rangle$. Here, s is the mapping from good hosts to their local stacks, R is a predicate on E indicating which entry points are running, and $T_R \subseteq T$ is the set of tokens released to bad hosts or generated by them. The intention is that T_R contains all tokens that have been passed to the bad hosts or to low integrity entry points before reaching this system configuration during the computation.

As always, we cannot assume anything about what code a bad host is running. Thus, we use the predicate R only for those entry points located on good hosts. The notation $R[e \mapsto x]$ for $x \in \{\mathit{true}, \mathit{false}\}$ stands for the predicate on entry points that agrees with R everywhere except e, for which $R[e \mapsto x](e) = x$.

The behavior of a Jif/split system can now be seen as a *labeled transition system* in which the states are system configurations and the transitions are actions. The notation

$$\langle s, R, T_R \rangle \xrightarrow{(h,a)} \langle s', R', T_R' \rangle$$

indicates that left configuration transitions via the action (h, a) to yield the right configuration. The effects of the transition on the configuration depend on the action. For example, a successful `lgoto` request will cause a good host h to pop its local token stack. Thus, for that transition $s_h' = \mathit{pop}(s_h)$. Other effects, such as passing a token to a bad host, relate T_R and T_R'. The proof cases in Section A.5 describe the effects of each transition.

Not every transition sequence is possible—some are ruled out by the static constraints, while some are ruled out by the dynamic checks of good hosts. Thus, we must make some further *modeling assumptions* about the valid transition systems.

- If, during a computation, the configuration S transitions to a configuration S' via an action performed by a good host, then that action is actually the result of evaluating an entry point on that host:

$$\langle s, R, T_R \rangle \xrightarrow{(h,a)} \langle s', R', T_R' \rangle \wedge h \in H_G \Rightarrow \exists e \in E_h, f, t \,.\, (e(f,t) = (h,a)) \wedge R(e)$$

- If the initiator of an action is a bad host or an entry point in E_B, then any tokens appearing in the action are available to the bad hosts (they are in the set T_R).

A.4 The Stack Integrity Invariant

This section defines the stack integrity invariant, which will establish the correctness of control-transfer protocol. First, we define some useful notation for describing the relevant properties of the system configurations.

Each token $t = \{h, f, e\}_{k_h}$ generated by a good host h corresponds to the entry point e. Because tokens are hashed with a private key, it is possible to distinguish tokens generated by good hosts from tokens generated by bad hosts. For any token t, let $\mathit{creator}(t)$ be the host that signed the token (in this case, host h). Using these pieces of information, we can define the integrity level of a token as:

$$\mathtt{I}_t = \begin{cases} \mathtt{I}_e & h = \mathit{creator}(t) \in H_G \\ \mathtt{I}_h & h = \mathit{creator}(t) \in H_B \end{cases}$$

Define a *good token* to be any token t for which $\mathtt{I}_t \sqsubseteq \iota$. Let T_G be the set of all good tokens and $T_B = T \setminus T_G$ be the set of bad tokens.

Just as we have partitioned hosts and tokens into good and bad sets, we define *good entry points* and *bad entry points*. The set of low integrity entry points can be defined as $E_G = \{e \mid e \in E \wedge \mathtt{I}_e \not\sqsubseteq \iota\}$. Correspondingly, let $E_B = E \setminus E_G$ be the set of high-integrity entry points.

Recall from Section 5 that the local stacks are intended to simulate a global integrity control stack (ICS) that corresponds to the program counter of the source program. Due to the presence of bad hosts, which may request `sync` operations with low-integrity entry points located at good hosts, the global structure described by the composition of the local stacks may not be a stack. To see why, consider a bad host that obtains a good token t and then uses the good token in `sync` requests to bad entry points on two different good hosts, h_1 and h_2. The resulting configuration of local stacks contains $s_{h_1} = \ldots : (t_1, t)$ and $s_{h_2} = \ldots : (t_2, t)$. Thus the global ICS isn't a stack, but a directed, acyclic graph. However, the crucial part of the invariant is that the global ICS is a stack with respect to good tokens.

The key to defining the invariant is to relax the notion of "stack" with respect to bad tokens. Observe that the global structure on T induced by the local stacks is captured by the directed graph whose nodes are tokens in T and whose edges are given by $\{(t, t') \mid \exists h \in H_G.(t, t') \in s_h\}$. If this structure truly described a stack there would be a single component:

$$t_1 \rightarrow t_2 \rightarrow \ldots t_{n-1} \rightarrow t_n$$

with no other edges present. (Here t_n is the bottom of the stack.) Instead, we show that the graph looks like:

$$B \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots t_{n-1} \rightarrow t_n$$

where B is an arbitrary dag whose nodes are only bad tokens and t_1 through t_n are good tokens.

We formalize the 'B' part of the graph by observing that the edges in B and the ones between B and t_1 originate from a bad token. Because the good hosts' local stacks change during a run of the system, it is convenient to define some notation that lets us talk about this property in a given configuration. Let S be a system configuration, $\langle s, R, T_R \rangle$. Define:

$$t \prec_S t' \quad \Leftrightarrow \quad \exists h \in H_G.(t, t') \in s_h \ \wedge \ t \in T_B$$

The relation $t \prec_S t'$ says that the bad token t appears immediately before the token t' in the graph above. Its transitive closure $t \prec_S^* t'$ says that there are bad tokens u_1 through u_n such that

$$t \rightarrow u_1 \rightarrow \ldots \rightarrow u_n \rightarrow t'$$

appears as part of the ICS—these bad tokens are a subgraph of the dag B. Note that t may be either a good or a bad token. The pseudo-stack property of the invariant (see (iv) below) says that there is at most one good token reachable through the relation $\prec_S^*$—that is, at most one good token can be a successor of B, the bad portion of the ICS.

If we conservatively assume that all bad tokens are available to the bad hosts, then $t \prec_S^* t'$ says that the bad hosts can "get to" the token t' by doing an appropriate series of `lgoto` operations (each of which will pop a t_i off the ICS).

We next define some auxiliary concepts needed to state the stack invariant:

$$\begin{array}{rcl} \mathit{Tok}_S & = & \{t \mid \exists h \in H_G . t \in \mathit{fst}(s_h)\} \\ \mathit{Tok}_S(h) & = & \mathit{fst}(s_h) \text{ whenever } h \in H_G \\ \mathit{Active}_S(t) & \Leftrightarrow & t \in \mathit{Tok}_S \wedge \exists t' \in T_R.(t = t') \vee (t' \prec_S^* t) \end{array}$$

The set Tok_S is just the set of tokens appearing in the left part of any good host's local stack—this is a set of tokens for which some good host might grant an `lgoto` request (it is conservative because it includes all the left-hand-side tokens of the stack, not just the top of the stack). The set $\mathit{Tok}_S(h)$ is the restriction of Tok_S to a particular good host h. $\mathit{Tok}_S(h)$ is exactly the set of tokens issued by h that have not been consumed by a valid `lgoto` request. Finally, the predicate $\mathit{Active}_S(t)$ determines the subset of Tok_S "reachable" from the tokens available to bad hosts.

Definition 1 *A configuration S satisfies the Stack Integrity Invariant (SII) if and only if:*

(i) $\forall t, t' \in T_G . \mathit{Active}_S(t) \wedge \mathit{Active}_S(t') \Rightarrow t = t'$

Uniqueness of exposed, good tokens.

(ii) $\exists e \in E_G .\ R(e) \Rightarrow \neg\exists t \in T_G . \mathit{Active}_S(t)$

When good code has control, there are no good, active tokens.

(iii) $\forall e, e' \in E_G .\ R(e) \wedge R(e') \Rightarrow e = e'$

Good code is single threaded.

(iv) $\forall t_1, t_1', t_2, t_2' .\ (t_1 \prec_S^* t_1') \wedge (t_2 \prec_S^* t_2') \wedge (t_1', t_2' \in T_G) \Rightarrow t_1' = t_2'$

Pseudo-linearity of stacks.

(v) $\forall h_1, h_2 \in H_G .\ h_1 \neq h_2 \Rightarrow \mathit{Tok}_S(h_1) \cap \mathit{Tok}_S(h_2) = \emptyset$.

No two good hosts generate identical tokens.

(vi) $\forall h \in H_G .\ s_h = (t_1, t_1') : \ldots (t_n, t_n') \Rightarrow$ *t_1 through t_n are pairwise distinct.*

Stack Integrity Theorem *If S is a configuration satisfying the SII and S transitions to S', then S' satisfies the SII.*

Note that condition (i) of the SII implies that if t is a good token on the top of some good host's local stack and t has been handed out as a capability to the bad hosts ($t \in T_R$), then t is unique—there are no other such capabilities available to the bad hosts. Because only good hosts can create such tokens, and they do so only by following the source program's control flow, the bad hosts cannot subvert the control-flow of high-integrity computation.

A.5 Proof of the Stack Integrity Theorem

Suppose $S = \langle s, R, T_R \rangle$ is a configuration satisfying the Stack Integrity Invariant (SII). To show that our system preserves the SII, we reason by cases on all the possible actions in the system. In other words, we want to show that after any possible action, the resulting system configuration $S' = \langle s', R', T_R' \rangle$ still satisfies SII. Note that any communication between bad hosts does not change the state of the configuration, so we may safely eliminate those cases. We first make a few observations:

1. If $s' = s$ then invariants (iv), (v), and (vi) hold trivially because they depend only on the state of the local stacks.

2. If $s' = s$ and $T'_R = T_R$ then $\mathit{Active}_{S'} = \mathit{Active}_S$ and invariant (i) is satisfied trivially.

3. Changing the running predicate of a bad entry point does not affect invariants (ii) or (iii)—changing the running predicate on good entries, or changing Active_S may affect (ii) and (iii).

Case I. S transitions via $(h_1, \texttt{rgoto}(h_2, f_2, e_2, t))$.

(a) $h_1 \in H_B$, $h_2 \in H_G$, and $I_{h_1} \not\sqsubseteq I_{e_2}$.
In this case, because h_2 is a good host, the dynamic check on `rgoto` prevents the system configuration from changing. Thus $S' = S$, and the invariant is immediate.

(b) $h_1 \in H_B$, $h_2 \in H_G$, and $I_{h_1} \sqsubseteq I_{e_2}$.
Because $h_1 \in H_B$, we have $I_{h_1} \not\sqsubseteq \iota$ and thus $I_{e_2} \not\sqsubseteq \iota$. Consequently, $e_2 \in E_B$. Thus $T'_R = T_R \cup \{t\} = T_R$, $R' = R[e_2 \mapsto \mathit{true}]$, and $s' = s$. Observations 1, 2, and 3 show that all of the invariants but (ii) hold for S'. Invariant (ii) follows from the fact that $\mathit{Active}_{S'} = \mathit{Active}_S$ and the fact that invariant (ii) holds in S.

(c) $h_1 \in H_G$ and $h_2 \in H_B$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \texttt{rgoto}(h_2, f_2, e_2, t))$ for some f_1 and, furthermore, $R(e_1)$ holds. In the new configuration, $T'_R = T_R \cup \{t\}$, $R' = R[e_1 \mapsto \mathit{false}]$, and $s' = s$. Observation 1 shows that invariants (iv), (v) and (vi) hold trivially because $s' = s$. Note that we also have $\prec^*_S = \prec^*_{S'}$.

If $e_1 \in E_B$ then $t \in T_R$, so $T'_R = T_R$ and observations 2 and 3 imply that S' satisfies the SII.

Otherwise $e_1 \in E_G$ and by invariant (iii) of state S no other good entry points are running. Thus, in S' we have $\forall e \in E_G. \neg R(e)$ and it follows that invariants (ii) and (iii) hold in S'. Now we must show that invariant (i) holds. Consider an arbitrary good token $u \in T_G$. Because $T'_R = T_R \cup \{t\}$ we have

$$\begin{aligned}
\mathit{Active}_{S'}(u) \quad &\Leftrightarrow \quad u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\
&\Leftrightarrow \quad u \in \mathit{Tok}_S \wedge \exists u' \in T_R \cup \{t\}.(u = u') \vee (u' \prec^*_S u) \\
&\Leftrightarrow \quad [t \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)] \vee \mathit{Active}_S(u)
\end{aligned}$$

By invariant (ii) of configuration S, we have $u \in T_G \Rightarrow \neg \mathit{Active}_S(u)$ and so $u \in T_G \wedge \mathit{Active}_{S'}(u)$ implies $t \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)$. If $t \in T_G$ then by the definition of $\prec^*_S$ we have $\neg \exists u. t \prec^*_S u$ and consequently $u \in T_G \wedge \mathit{Active}_{S'}(u) \Rightarrow u = t$, from which invariant (i) follows immediately. Otherwise, $t \in T_B$ and we have that $u_1, u_2 \in T_G \wedge \mathit{Active}_{S'}(u_1) \wedge \mathit{Active}_{S'}(u_2) \Rightarrow t \prec^*_S u_1 \wedge t \prec^*_S u_2$, but now invariant (iv) of configuration S implies that $u_1 = u_2$ as needed. Thus invariant (i) holds in S'.

(d) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \not\sqsubseteq \mathtt{I}_{e_2}$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \texttt{rgoto}(h_2, f_2, e_2, t))$ for some frame f_1 and, furthermore, $R(e_1)$ holds. After the transition, we have $R' = R[e_1 \mapsto \mathit{false}]$, and, due to the run-time check performed by h_2, we also have $T'_R = T_R$, and $s' = s$. Invariants (i), (iv), (v) and (vi) follow immediately from observations 1 and 2. If $e_1 \in E_B$ then invariants (ii) and (iii) are a direct consequence of the assumption that they hold in S. To see that (ii) and (iii) hold when $e_1 \in E_G$, note that because $R(e_1)$ we have $\forall e \in E_G. R(e) \Rightarrow e = e_1$ (from invariant (iii)). Because R' agrees with R everywhere but e_1, (iii) holds of R' too. The same reasoning shows that (ii) holds.

(e) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \sqsubseteq \mathtt{I}_{e_2}$.

By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \texttt{rgoto}(h_2, f_2, e_2, t))$ for some frame f_1 and, furthermore, $R(e_1)$ holds. After the transition, we have

$$R' = R[e_1 \mapsto \mathit{false}][e_2 \mapsto \mathit{true}]$$

and $s' = s$. This latter fact and observation 1 implies that invariants (iv), (v), and (vi) hold in S'. Note also that $\prec^*_{S'} = \prec^*_S$.

If $e_1 \in E_B$, the modeling assumption tells us that $T'_R = T_R \cup \{t\} = T_R$ because $t \in T_R$. Note that because h_1 is a good host, the static constraint on $\texttt{rgoto}$ implies that $\texttt{I}_{e_1} \sqsubseteq \texttt{I}_{e_2}$, which in turn implies that $\texttt{I}_{e_2} \not\sqsubseteq \iota$ and thus $e_2 \in E_B$. Invariants (i), (ii), and (iii) follow immediately from observations 2 and 3, plus the fact that R' agrees with R on all good entry points.

Otherwise, $e_2 \in E_G$. If $e_1 \in E_G$ then $T'_R = T_R$ because t is not passed to a bad entry point. Consequently, $\mathit{Active}_{S'} = \mathit{Active}_S$ and invariant (i) follows immediately. Because $R(e_1) \wedge e_1 \in E_G$, invariant (iii) of S implies that no other good entry points are running in predicate R. Thus, because $R' = R[e_1 \mapsto \mathit{false}][e_2 \mapsto \mathit{true}]$ it is trivial to show that $R'(e) \wedge e \in E_G \Rightarrow e = e_2$, as required. Furthermore, $R(e_1)$ implies that $\neg\exists t \in T_G.\mathit{Active}_S(t)$ and so invariant (ii) holds in configuration S' too.

The last case is when $e_1 \in E_G$ and $e_2 \in E_B$, but this case follows exactly as in the last paragraph of case I(c).

Case II. S transitions via $(h_1, \texttt{lgoto}(t))$ where $t = \{h_2, f_2, e_2\}_{k_{h_2}}$.

(a) $h_1 \in H_B, h_2 \in H_G$ and $\mathit{top}(s_{h_2}) \neq (t, t')$.
In this case, because h_2 is a good host, the dynamic check on $\texttt{lgoto}$ prevents the system configuration from changing. Thus $S' = S$, and the invariant is immediate.

(b) $h_1 \in H_B, h_2 \in H_G$ and $\mathit{top}(s_{h_2}) = (t, t')$ for some token t'.
Note that $t \in \mathit{Tok}_S$, and by the modeling assumption, $t \in T_R$ and, consequently, we have $\mathit{Active}_S(t)$. Because h_2 pops its local stack, invariants (v) and (vi) of configuration S imply that $\mathit{Tok}_{S'}(h_2) = \mathit{Tok}_S(h_2) \setminus \{t\}$ and thus $\mathit{Tok}_{S'} = \mathit{Tok}_S \setminus \{t\}$. Also note that because of the stack pop $\prec^*_{S'} \subseteq \prec^*_S$, which implies that SII(iv) holds in configurations S'. Invariants (v) and (vi) hold in S' directly because they hold in S. There are two final cases to consider:

1. $t \in T_G$
It follows that $e_2 \in E_G$, and thus $T'_R = T_R$. Furthermore, $R' = R[e_2 \mapsto \mathit{true}]$. We now show that $\mathit{Active}_{S'}(u) \Rightarrow \mathit{Active}_S(u)$:

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in (\mathit{Tok}_S \setminus \{t\}) \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Rightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_S u) \\ & \Leftrightarrow & \mathit{Active}_S(u) \end{array}$$

We show that in S' it is the case that $\forall u \in T_G.\neg\mathit{Active}_{S'}(u)$, from which invariants (i) and (ii) follow directly. Suppose for the sake of contradiction that $\mathit{Active}_{S'}(u)$ for some $u \in T_G$. Then, by the implication above, we have $\mathit{Active}_S(u)$. Recall that $\mathit{Active}_S(t)$, and so by invariant (ii) of the configuration S, we have $u = t$. But, $\mathit{Active}_{S'}(u) \Rightarrow u \in \mathit{Tok}_{S'} = \mathit{Tok}_S \setminus \{t\}$, which implies that $u \neq t$, a contradiction.

Lastly, we must show that SII(iii) holds in configuration S'. We know that $R'(e_2) = \mathit{true}$. Suppose $e \in E_G$ and assume $e \neq e_2$. We must show that $\neg R'(e)$. But, $R'(e) = R[e_2 \mapsto \mathit{true}](e) = R(e)$. Recalling once more that $\mathit{Active}_S(t) \wedge t \in T_G$, the contrapositive of SII(ii) for configuration S implies that $\neg R(e)$ as desired.

2. $t \in T_B$
   It follows that $e_2 \in E_B$, and thus $T'_R = T_R \cup \{t'\}$. Furthermore, $R' = R[e_2 \mapsto \mathit{true}]$ and we immediately obtain invariant (iii) via observation 3. First note that $t \prec_S t'$ because $(t, t') \in s_{h_2}$, and, consequently, if $t' \in T_B$ we have $t \prec^*_S u \wedge u \neq t' \Rightarrow t' \prec^*_S u$ for any u. We need this fact to derive the implication marked $\star$ below:

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in (\mathit{Tok}_S \setminus \{t\}) \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec^*_{S'} u) \\ \star & \Rightarrow & u \in \mathit{Tok}_S \wedge ((u = t') \vee \exists u' \in T_R.(u = u') \vee (u' \prec^*_{S'} u)) \\ & \Leftrightarrow & (u \in \mathit{Tok}_S \wedge u = t') \vee \mathit{Active}_S(u) \end{array}$$

   If $t' \in \mathit{Tok}_S$, then by definition, we have $\mathit{Active}_S(t')$; otherwise in the left conjunct above we have $(u \in \mathit{Tok}_S \wedge u = t' \wedge t' \notin \mathit{Tok}_S) = \mathit{false}$. Thus, in either case, the expression above reduces to $\mathit{Active}_S(u)$ and we have $\mathit{Active}_{S'}(u) \Rightarrow \mathit{Active}_S(u)$. Invariant (i)in S' follows directly from invariant (i) of S; similarly because R' agrees with R on good entry points, invariant (ii) in S' follows directly from invariant (ii) of S.

(c) $h_1 \in H_G$ and $h_2 \in H_G$, and $\mathit{top}(s_{h_2}) \neq (t, t')$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \texttt{lgoto}(t))$ for some f_1 and, furthermore, $R(e_1)$ holds. Because $h_1 \in H_G$, we have $R' = R[e_1 \mapsto \mathit{false}]$, but the static checks performed by good host h_2 imply that $s' = s$ and $T'_R = T_R$. Invariant (ii) follows from the facts that $R'(e) \Rightarrow R(e)$ and $\mathit{Active}_{S'} = \mathit{Active}_S$. The rest of the invariants follow directly from observations 1,2,and 3.

(d) $h_1 \in H_G$ and $h_2 \in H_G$, and $\mathit{top}(s_{h_2}) = (t, t')$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \texttt{lgoto}(t))$ for some f_1 and, furthermore, $R(e_1)$ holds. Because $h_1 \in H_G$, we have $R' = R[e_1 \mapsto \mathit{false}][e_2 \mapsto \mathit{true}]$. Note that invariants (iv), (v) and (vi) for S' follow directly from the same invariants of S; popping s_{h_2} implies that $\prec^*_{S'} \subseteq \prec^*_S$.
If $e_1 \in E_B$ then $t \in T_R$ and we use the same reasoning as in Case II.(b).
Otherwise, $e_1 \in E_G$. Note that invariant (ii) of configuration S implies that $\neg\exists u \in T_G.\mathit{Active}_S(u)$ and invariant (iii) implies that $e \in E_G \wedge R(e) \Rightarrow e = e_1$.

1. $t \in T_G$.
   In this case, $e_2 \in E_G$. We first show that invariant (iii) holds in S'. We know that $R'(e_2) = \mathit{true}$, so let $e \in E_G$ be given. We must show that $R'(e) \Rightarrow e = e_2$. Suppose for the sake of contradiction that $R'(e)$ and $e \neq e_2$ then

$$R'(e) = R[e_1 \mapsto \mathit{false}][e_2 \mapsto \mathit{true}](e) = R[e_1 \mapsto \mathit{false}](e) \wedge R'(e) \Rightarrow e \neq e_1$$

   But this contradicts invariant (iii) of configuration S which says that $e \in E_G \wedge R(e) \wedge R(e_1) \Rightarrow e = e_1$. We conclude that $e = e_2$ as desired.

   Next, we show that $\mathit{Active}_{S'}(u) \Rightarrow \mathit{Active}_S$:

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in (\mathit{Tok}_S \setminus \{t\}) \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Rightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_S u) \\ & \Leftrightarrow & \mathit{Active}_S(u) \end{array}$$

   From this implication and the fact that $R(e_1)$ holds, we use invariant (ii) to conclude that $\neg\exists t \in T_G.\mathit{Active}_{S'}(t)$. Consequently, S' satisfies invariants (i) and (ii) as required.

2. $t \in T_B$.

   In this case, $e_2 \in E_B$ and it follows that $e_2 \neq e_1$. We show that there are no good, running entry points in S': Let $e \in E_G$ be given. We immediately have that $e \neq e_2$. If $e = e_1$, then as required:

$$R'(e) = R[e_1 \mapsto \textit{false}][e_2 \mapsto \textit{true}](e) = R[e_1 \mapsto \textit{false}](e) = \textit{false}.$$

   Assuming $e \neq e_1$ we have $R'(e) = R(e)$, and by invariant (iii) of configuration S it follows that $R(e) = \textit{false}$. Thus, invariants (ii) and (iii) of configurations S' hold trivially.

   To show invariant (i), note that $T'_R = T_R \cup \{t'\}$.

$$\begin{array}{rcl} \textit{Active}_{S'}(u) & \Leftrightarrow & u \in \textit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in \textit{Tok}_S \setminus \{t\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Rightarrow & u \in \textit{Tok}_S \wedge ((u = t') \vee (t' \prec^*_{S'} u) \vee \exists u' \in T_R.(u = u') \vee (u' \prec^*_S u)) \\ & \Rightarrow & (u \in \textit{Tok}_S \wedge ((u = t') \vee (t' \prec^*_{S'} u))) \vee \textit{Active}_S(u) \end{array}$$

   Let $u, u' \in T_G$ be given and suppose $\textit{Active}_{S'}(u) \wedge \textit{Active}_{S'}(u')$. Note that invariant (ii) of configuration S implies that $\neg\exists u \in T_G.\textit{Active}_S(u)$, thus we have $\textit{Active}_{S'}(u) \Rightarrow (u \in \textit{Tok}_S \wedge (u = t') \vee (t' \prec^*_{S'} u))$ and similarly, $\textit{Active}_{S'}(u') \Rightarrow (u' \in \textit{Tok}_S \wedge (u' = t') \vee (t' \prec^*_{S'} u'))$. Suppose $u = t'$. Then $t' \in T_G$ and from the definition of $\prec^*_{S'}$ it follows that $\neg(t' \prec^*_{S'} u')$ which implies that $u' = t' = u$ as required. Otherwise, we have $t' \prec^*_{S'} u$, which means that $t' \in T_B$ and it follows that $t' \prec^*_{S'} u'$. But this implies $t' \prec^*_S u \wedge t' \prec^*_S u'$, so by invariant (iv) of configuration S, we have $u = u'$.

(e) $h_1 \in H_G$ and $h_2 \in H_B$.

By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \texttt{lgoto}(t))$ for some f_1 and, furthermore, $R(e_1)$ holds. Because $h_1 \in H_G$, we have $R' = R[e_1 \mapsto \textit{false}]$. Because host $h_2 \in H_B$ we have $s' = s$ and $T'_R = T_R$. Invariant (ii) follows from the facts that $R'(e) \Rightarrow R(e)$ and $\textit{Active}_{S'} = \textit{Active}_S$. The rest of the invariants follow directly from observations 1,2,and 3.

Case III. S transitions via $(h_1, \texttt{sync}(h_2, f_2, e_2, t))$.

(a) $h_1 \in H_B$ and $h_2 \in H_G$ and $\texttt{I}_{h_1} \not\sqsubseteq \texttt{I}_{e_2}$.

In this case, because h_2 is a good host, the dynamic check on `rgoto` prevents the system configuration from changing. Thus $S' = S$, and the invariant is immediate.

(b) $h_1 \in H_B$ and $h_2 \in H_G$ and $\texttt{I}_{h_1} \sqsubseteq \texttt{I}_{e_2}$.

Because $h_2 \in H_G$, we have $s'_{h_2} = s_{h_2} : (t', t)$ where $t' = \{h_2, f_2, e_2\}_{k_{h_2}}$ is a fresh token. Invariants (v) and (vi) hold in S' because they hold in S and t' is fresh. Furthermore, because $\texttt{I}_{h_1} \sqsubseteq \texttt{I}_{e_2} \wedge h_1 \in H_B$ it follows that $\texttt{I}_{e_2} \not\sqsubseteq \iota$, and consequently $t' \in T_B$. $R' = R$ because no good host begins running after this transition; invariant (iii) follows directly.

We next show that invariant (iv) is met. Observe that $\prec_{S'} = \prec_S \cup \{(t', t)\}$. In particular, $\neg\exists u. u \prec_{S'} t'$ and so we have

$$u \prec^*_{S'} u' \Leftrightarrow (u \prec^*_S u') \vee (u = t' \wedge t \prec^*_S u')$$

Let u_1, u'_1, u_2, u'_2 be given and suppose that $(u_1 \prec^*_{S'} u'_1) \wedge (u_2 \prec^*_{S'} u'_2) \wedge (u'_1, u'_2 \in T_G)$. From the definition of $\prec^*_{S'}$ we obtain:

$$[(u_1 \prec^*_S u'_1) \vee (u_1 = t' \wedge t \prec^*_S u'_1)] \;\wedge\; [(u_2 \prec^*_S u'_2) \vee (u_2 = t' \wedge t \prec^*_S u'_2)]$$

But for each of the four possible alternatives described above, invariant (iv) of configuration S implies that $u'_1 = u'_2$ as needed. For example, if $(u_1 \prec^*_S u'_1) \wedge (t \prec^*_S u'_2)$ then instantiating (iv) with $t_1 = u_1, t'_1 = u'_1, t_2 = t, t'_2 = u'_2$ yields $u'_1 = u'_2$. The other cases are similar.

Next we show that invariants (i) and (ii) are maintained. First, note that $T'_R = T_R \cup \{t'\}$ because h_2 sends the fresh token to h_1. Also observe that $\mathit{Tok}_{S'} = \mathit{Tok}_S \cup \{t'\}$ because h_2 has pushed (t', t) onto its local stack. We use the fact that $t \in T_R$ in the derivation marked $\star$ below:

$$\begin{array}{rrl}
\mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec^*_{S'} u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec^*_S u \vee (u' = t' \wedge t \prec^*_S u)) \\
\star & \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec^*_S u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge (u = t' \vee \exists u' \in T_R.(u = u') \vee (u' \prec^*_S u)) \\
& \Leftrightarrow & u = t' \vee \mathit{Active}_S(u)
\end{array}$$

Note that, because $t' \in T_B$, we have $\mathit{Active}_{S'}(u) \wedge u \in T_G \Rightarrow \mathit{Active}_S(u)$. Consequently, invariants (i) and (ii) hold in S' because they hold in S.

(c) $h_1 \in H_G$ and $h_2 \in H_B$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \texttt{sync}(h_2, f_2, e_2, t))$ for some frame f_1. Furthermore, $R(e_1)$ holds. After this transition, $s' = s$, and $T'_R = T_R \cup \{t\}$ because t has been passed to a bad host. Observation 1 shows that invariants (iv), (v) and (vi) hold immediately. The new running predicate is:

$$R' = R[e_1 \mapsto \mathit{false}][\mathit{send}(h_2)_{h_1} \mapsto x]$$

Where x can be either *true* or *false*, depending on whether the bad host h_2 replies with a token to h_1. However, because h_1 is a good host, the static constraints on inserting $\texttt{sync}$'s imply that $\mathtt{I}_{h_2} \sqsubseteq I(\texttt{pc})$. But then, because $h_2 \in H_B$, it follows that $\mathtt{I}_{h_2} \not\sqsubseteq \iota \Rightarrow I(\texttt{pc}) \not\sqsubseteq \iota$. Furthermore, because the integrity label on the $\mathit{send}(h_2)_{h_1}$ entry point is just $I(\texttt{pc})$, we have that $\mathit{send}(h_2)_{h_1} \in E_B$. Thus, whether $\mathit{send}(h_2)_{h_1}$ is running does not affect invariants (ii) and (iii).
Next we calculate the predicate $\mathit{Active}_{S'}$, recalling that $T'_R = T_R \cup \{t\}$:

$$\begin{array}{rcl}
\mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R \cup \{t\}.(u = u') \vee (u' \prec^*_S u) \\
& \Leftrightarrow & (u \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)) \vee \mathit{Active}_S(u)
\end{array}$$

1. $e_1 \in E_G$.
   In this case, because $R(e_1)$ holds in configuration S, invariant (ii) tells us that when $u \in T_G$ and $\mathit{Active}_{S'}(u)$ it is the case that $(u \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u))$. To show that invariant (i) holds in S', suppose $u, u' \in T_G \wedge \mathit{Active}_{S'}(u) \wedge \mathit{Active}_{S'}(u')$. Then we have

   $$[u \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)] \;\wedge\; [u' \in \mathit{Tok}_S \wedge (u' = t \vee t \prec^*_S u')]$$

   Now suppose $t \in T_G$. Then by definition $\neg\exists t'.t \prec^*_S t'$, so the above condition on u and u' becomes $u, u' \in \mathit{Tok}_S \wedge u = t \wedge u' = t$ as required. Otherwise, if $t \in T_B$, it follows that $u \neq t$ and $u' \neq t$ and we have $t \prec^*_S u$ and $t \prec^*_S u'$. But then invariant (iv) of configuration S implies that $u = u'$ in this case as well.

   To show that invariants (ii) and (iii) hold, we prove that $\neg\exists e \in E_G.R'(e)$. Suppose for contradiction that there was such an e. By the definition of R', we conclude that $e \neq e_1$, but then $R'(e) = R(e)$. From the modeling assumption, we have $R(e_1)$, and yet invariant (iii) of configuration S implies that $e = e_1$, a contradiction.
2. $e_1 \in E_B$.
   In this case, the modeling assumption tells us that $t \in T_R$, so $T'_R = T_R$. This fact immediately yields that $\mathit{Active}_{S'} = \mathit{Active}_S$, and observations 2 and 3, imply (i) and (iii) hold in S'. Invariant (ii) in S' also follows directly from invariant (ii) in S.

(d) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \not\sqsubseteq \mathtt{I}_{e_2}$.
This case is identical to I.(d).

(e) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \sqsubseteq \mathtt{I}_{e_2}$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that $e_1(f_1, t) = (h_1, \mathtt{sync}(h_2, f_2, e_2, t))$ for some frame f_1 and, furthermore, $R(e_1)$ holds. Because $h_2 \in H_G$, we have $s'_{h_2} = s_{h_2} : (t', t)$ where $t' = \{h_2, f_2, e_2\}_{k_{h_2}}$ is a fresh token. Invariants (v) and (vi) hold in S' because they hold in S and t' is fresh. After the transition, we have

$$R' = R[e_1 \mapsto \mathit{false}][\mathit{send}(h_2)_{h_1} \mapsto \mathit{true}]$$

1. $e_2 \in E_G$
In this case, $t' \in T_G$. It follows that $\neg\exists u.t' \prec_{S'} u$, and consequently $\prec^*_{S'} = \prec^*_S$, from which we conclude that invariant (iv) holds in S'. Because $h_1 \in H_G$ the static constraints on sync guarantee that $\mathtt{I}_{e_1} \sqsubseteq \mathtt{I}_{e_2}$, which implies that $e_1 \in E_G$.

If $\mathit{send}(h_1)_{h_2} \in E_G$ then $T'_R = T_R$. We now show that $\forall e \in E_G.R'(e) \Rightarrow e = \mathit{send}(h_1)_{h_2}$, from which we may immediately derive that invariant (iii) holds in S'. Let $e \in E_G$ be given and suppose for the sake of contradiction that $R'(e) \wedge e \neq \mathit{send}(h_1)_{h_2}$. From the definition of R' we have $R'(e) = R[e_1 \mapsto \mathit{false}](e)$, and because $R'(e)$ holds, we have that $e \neq e_1$. Thus $R'(e) = R(e)$. But now invariant (iii) and the assumption that $R(e_1)$ holds in S imply that $e = e_1$, a contradiction. Note that $T'_R = T_R \wedge s' = s \Rightarrow \mathit{Active}_{S'} = \mathit{Active}_S$. Invariants (i) and (ii) follow directly from the fact that they hold in configuration S.

Otherwise, $\mathit{send}(h_1)_{h_2} \in E_B$ and $T'_R = T_R \cup \{t'\}$. We first show that $\forall e \in E_G.\neg R'(e)$, from which invariants (ii) and (iii) follow immediately. Let $e \in E_G$ be given. We know that $e \neq \mathit{send}(h_1)_{h_2}$ because $\mathit{send}(h_1)_{h_2} \in E_B$; thus from the definition of R' we obtain $R'(e) = R[e_1 \mapsto \mathit{false}](e)$. If $e_1 = e$ we are done. So we have that $e_1 \neq e$. Now, however, $R(e) \wedge (e \in E_G)$ implies via invariant (iii) of configuration S that $e = e_1$, a contradiction. Thus we conclude that $\neg R(e)$ as desired.
It remains to show that invariant (i) holds in S', so we calculate the predicate $\mathit{Active}_{S'}$. The step marked $\star$ below uses the fact that $t' \in T_G$ (which, from the definition of $\prec_{S'}$ implies that $\neg\exists u.t' \prec^*_{S'} u$):

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u' = u) \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u' = u) \vee (u' \prec^*_S u) \\ \star & \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge [(u = t') \vee \exists u' \in T_R.(u' = u) \vee (u' \prec^*_S u)] \\ & \Rightarrow & (u = t') \vee \mathit{Active}_S(u) \end{array}$$

Observe, however, that invariant (ii) of configuration S implies that $\neg\exists u \in T_G.\mathit{Active}_S(u)$. Thus, $\mathit{Active}_{S'}(u) \wedge u \in T_G \Rightarrow u = t'$, and invariant (i) of configuration S' follows directly.

2. $e_2 \in E_B$
In this case, $t' \in T_B$. We use exactly the same reasoning as in Case III(b). to prove that invariant (iv) holds. Note that because $h_1 \in H_G$, the static constraints on sync imply that $e_1 \in E_G \Leftrightarrow \mathit{send}(h_1)_{h_2} \in E_G$.

In the case that $e_1 \in E_B$, we reason as in Case III(b). to show that invariants (i), (ii), and (iii) hold.

The last possibility is that $e_1 \in E_G$. Here, we have $T'_R = T_R$ because t' has not been passed to a bad entry point. Thus we can calculate $\mathit{Active}_{S'}$ as follows:

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R.(u = u') \vee [(u' = t' \wedge t' \prec^*_S u) \vee u' \prec^*_S u] \\ \star & \Leftrightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R.(u = u') \vee u' \prec^*_S u \\ & \Leftrightarrow & \mathit{Active}_S(u) \end{array}$$

In the reasoning above, the step marked $\star$ uses the fact that $t' \notin T_R \wedge \neg\exists u'.u' \prec^*_{S'} t'$. Invariant (i) follows directly from the fact that (i) holds in configuration S and the equivalence of $\mathit{Active}_{S'}$ and Active_S. To see that (ii) holds, note that $R(e_1) \Rightarrow \neg\exists t \in T_G.\mathit{Active}_S(t)$, but this implies $\neg\exists t \in T_G.\mathit{Active}'_S(t)$ as required. To establish invariant (iii), let $e \in E_G$ be given and suppose $R(e)$. We show that $e = \mathit{send}(h_1)_{h_2}$. Suppose by way of contradiction that $e \neq \mathit{send}(h_1)_{h_2}$. Then from the definition of R' we have $R'(e) = R[e_1 \mapsto \mathit{false}](e)$. By assumption $R'(e) = \mathit{true}$ and it follows that $e \neq e_1$. But now $R'(e) = R(e)$ and invariant (iii) shows that $e = e_1$, a contradiction.