Computer Sciences Department

DIFC Programs by Automatic Instrumentation

William R. Harris
Somesh Jha
Thomas Reps

Technical Report #1673

April 2010

DIFC Programs by Automatic Instrumentation

William R. Harris Somesh Jha Thomas Reps *

University of Wisconsin
{wrharris, jha, reps}@cs.wisc.edu

Abstract

Decentralized information flow control (DIFC) operating systems provide applications with mechanisms for enforcing information-flow policies for their data. However, significant obstacles keep such operating systems from achieving widespread adoption. One key obstacle is that DIFC operating systems provide only low-level mechanisms for allowing application programmers to enforce their desired policies. It can be difficult for the programmer to ensure that their use of these mechanisms enforces their high-level policies, while at the same time not breaking the underlying functionality of the application. These are issues both for programmers who would develop new applications for a DIFC operating system and for programmers who would port existing applications to a DIFC operating system.

Our work significantly eases this task. We present an automatic technique that takes as input a program with no DIFC code, and two policies: one that specifies prohibited information flows and one that specifies flows that must be allowed. Our technique then produces a new version of the input program that satisfies the two policies. To evaluate our technique, we created an automatic tool, called SWIM (for **S**ecure **W**hat **I** **M**ean), that implements the technique, and applied it to a set of real-world programs and policies. The results of our evaluation demonstrate that the technique is both sufficiently expressive to generate code for real-world policies, and that it can generate such code efficiently. It thus represents a significant contribution towards developing systems with strong end-to-end information-flow guarantees.

1. Introduction

Decentralized information flow control (DIFC) operating systems provide applications with mechanisms for ensuring the secrecy and integrity of their data [12, 19, 21] throughout a system. To enable this, a DIFC OS maps the set of executing processes to a partially-ordered set of *labels*. A process may send data to another process only if the processes' labels satisfy a certain ordering. A process may change its own labels and the labels of other processes, under restrictions enforced by the DIFC OS. Thus processes express how they intend their information to be shared by associating the labels with information, and the DIFC OS respects these intentions by enforcing a semantics of labels.

Previous work has concerned how to implement DIFC systems, in some cases atop standard operating systems. Furthermore, some systems have formal proofs that if an application running on the system correctly manipulates labels to implement a policy, then the system will enforce the policy [11]. However, for a user to have end-to-end assurance that their application implements a high-level information-flow policy, they must have assurance that the application indeed correctly manipulates labels. The label manipulations allowed by DIFC systems are expressive but low-level, so a high-level policy is semantically distant from a program's label manipulations. For the remainder of this paper, we narrow our discussion from general DIFC systems to Flume [12]. In principle, our approach can be applied to arbitrary DIFC operating systems. However, targeting Flume yields both theoretical and practical benefits. From a theoretical standpoint, Flume defines the semantics of manipulating labels in terms of set operations, a well-understood formalism. From a practical standpoint, Flume runs on Linux, giving our approach wide applicability. The work in [12] gives a comprehensive description of the Flume system.

```
void ap_mpm_run()
A1: while (*)
A2:   Conn c = get_request_connection();
A3:   Conn c’ = c; c = fresh_connection();
A4:   tag_t t = create_tag();
A5:   spawn(‘‘proxy’’, {c, c’},  {t}, {t}, {t} );
A6:   spawn(‘‘proxy’’, {c’, c},  {t}, {t}, {t} );
A7:   spawn(‘‘worker’’, c,  {t}, {t}, {} );

void proxy(Conn c, Conn c’,
        Label lab, Label pos_cap, Label neg_cap )
P1: while (*)
P2:   expand_label(pos_cap);
P3:   Buffer b = read(c);
P4:   clear_label(neg_cap);
P5:   write(c’, b);
```

Figure 1. An example derived from the Apache multi-process module.

To illustrate the gap between low-level label manipulations and high-level policies, consider the server program in Fig. 1, a simplified excerpt from an Apache multi-process module (MPM) [1]. For now, suppose that the program code consists only of the non-highlighted code. In this program, an MPM process executes the function ap_mpm_run, iterating through the loop indefinitely (lines A1 - A7). On each iteration of the loop, the MPM waits for a new connection c that communicates information for a service request (line A2). When the MPM receives a connection, it spawns a Worker process to handle the request (line A7). One desirable property for such a server is to isolate each Worker process: no Worker process should be able leak information to another Worker process, even if both processes are compromised and acting in collusion. It is difficult to design a server that upholds such a property: a Worker process may be compromised through any of a myriad of vulnerabilities, such as a stack-smashing attack. Once processes are compromised, they can communicate through any of several channels on the system, such as the file system.

However, the server can isolate Worker processes when executed on Flume. Suppose that the server in Fig. 1 is rewritten to include the highlighted code, which makes use of an API provided by the Flume. At a high level, Flume maps each process to three sets of *tags*, or atomic elements: the process's secrecy *label*, its *positive*

[*] Also affiliated with GrammaTech, Inc.

capability, and its *negative capability*. [1] A process may only alter its label by adding to it tags in its positive capability, or removing from it tags in its negative capability. In the highlighted code, the server uses the Flume API to ensure process isolation as follows. As before, an MPM process executes the function `ap_mpm_run`, and iterates through the loop indefinitely. Each time through the loop, it waits for a connection to a service request. Now, however, before spawning a Worker to handle the request, the MPM process performs two additional tasks. At line A4, it creates a fresh tag, and then initializes the label of the next Worker process with the tag, but does not give the Worker the capability to add or remove the tag from its label (line A7). This is the key to isolating the Worker processes. The Flume reference monitor intercepts all communications between processes, and only allows a communication to succeed if the label of the sending process is a subset of the label of the receiving process. Because the label of each Worker now contains a distinct tag that the Worker cannot remove, no Worker can send data to another. Flume associates other system objects, such as files, with tags as well, so the processes cannot communicate through files either.

However, while the label mechanisms provided by Flume are powerful, their power can lead to unintended side effects. In Fig. 1, the MPM process may not be able to alter the label of a process that issues service requests. If the MPM gave each Worker a unique tag and manipulated labels in no other way, then the label of each Worker would contain a tag t, while the label of the process that issued the request would not contain t. Thus the label of the Worker would not be a subset of the label of the receiver, and the Worker could not send information to the Requester. To resolve this, along with each Worker, the MPM process spawns two processes to serve as proxies for when the Worker receives information from (line A5) and sends information to (line A6) the Requester. Let Proxy_S be the process forwarding information from its associated Worker to the service requester. The MPM gives to Proxy_S the capability to add and remove from its label the tag t. To receive data from the Worker, Proxy_s expands its label to include t; to forward the data to the requester, it clears its label to be empty. Thus the untrusted Worker is isolated, while the small and trusted Proxy successfully communicates information between Worker and Requester.

This example illustrates that labels are a powerful mechanism for enabling application programmers to enforce information-flow policies. However, the example also illustrates that there is a significant gap between the high-level policies of programmers, e.g., that no Worker should be able to send information to another Worker, and the manipulations of labels required to implement such a policy. It also illustrates that these manipulations may be quite subtle when balancing desired security goals with the required functionality of an application. If programmers are to develop applications for DIFC systems, then they must resolve these issues manually. If they wish to instrument existing applications to run on DIFC systems, then they must ensure that their instrumentation implements their policy while not breaking the functionality requirements of an existing, and potentially large and complex, program. If their instrumentations do break functionality, it can be extremely difficult to discover this through testing: when a DIFC system blocks a communication, it may not report the failure, because such a report can leak information [12]. Label-manipulation code could thus introduce a new class of security and functionality bugs that could prove extremely difficult even to observe.

We have addressed this problem by creating an automatic *DIFC instrumenter*, which produces programs that, by construction, satisfy a high-level DIFC policy. Our DIFC instrumenter takes as input a program with no DIFC code, and two high-level declarative policies: one that specifies prohibited information flows (e.g. "Workers should not communicate with each other."), and one that specifies flows that must be allowed (e.g. "A Worker should be able to communicate to a proxy."). When successful, the instrumenter rewrites the input program with label-manipulation code so that it enforces the input policy. When unsuccessful, the instrumenter produces a minimal subprogram of the original program and a minimal subset of policies for which it could not find an instrumentation. To do so, the instrumenter reduces the problem of correctly instrumenting the program to a problem of solving a system of set constraints. It feeds the resulting constraint system to an off-the-shelf Satisfiability Modulo Theories (SMT) solver [7], which in our experiments found solutions in seconds (cf. Tab. 4). From a solution, the instrumenter instruments the program. Thus the programmer reasons at the policy level about information flow, and leaves to the instrumenter the task of correctly manipulating labels. If the programmer provides as input to the instrumenter the program in Fig. 1 minus the statements that manipulate labels, and a formal statement of a high-level policy similar to the one stated above, the instrumenter produces the entire program given in Fig. 1.

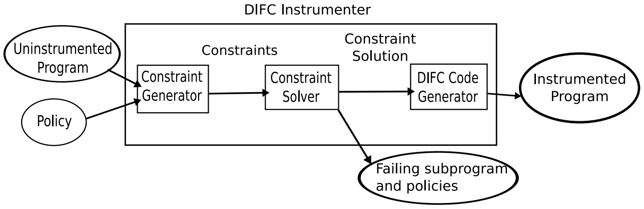

Figure 2. Workflow of the DIFC instrumenter.

The remainder of this paper is organized as follows: §2 gives an overview of our technique by describing the steps that it takes to instrument the example in Fig. 1. §3 formally describes the technique. §4 reports our experience applying the technique to real-world programs and information-flow policies. §5 places our work in the context of other work on DIFC systems and program synthesis. §6 concludes. Some technical details are covered in the Appendix.

2. Overview

We now informally describe each step of the workflow of the DIFC instrumenter using the example from Fig. 1. Fig. 2 illustrates the workflow.

2.1 Programs and Policies as Inputs

The instrumenter takes two inputs. First, it takes a program containing no Flume code. For the example in Fig. 1, the instrumenter takes the version of the server uninstrumented with calls to `create_tag()`, `expand_label()`, and `clear_label()`. The instrumenter analyzes programs represented as Communicating Sequential Processes (CSP) [2], but can automatically translate a program written in an imperative programing language, such as C, into a CSP program that models properties relevant for DIFC instrumentation.

Exa. 1. *The instrumenter translates the Apache server introduced in Fig. 1 to the following CSP program:*

$$\begin{array}{rclcrcl}
A_1 & = & A_5 & & P_1 & = & P_3 \\
A_5 & = & A_6 \mid\mid\mid P_1 & & P_3 & = & ?r \rightarrow P_5 \\
A_6 & = & A_7 \mid\mid\mid P_1 & & P_5 & = & !s \rightarrow P_1 \\
A_7 & = & A_1 \mid\mid\mid W & & \mathsf{init} & = & A_1 \mid\mid\mid R
\end{array}$$

[1] Flume also allows processes to protect their integrity via an integrity label. In this work, we focus on secrecy, but our techniques can be extended to reason about integrity. See App. D.

In the equations of the CSP program, each variable corresponds to a program point in the imperative program, and each equation defines the behavior of the program at the corresponding point. The expressions on the right-hand sides of equations are referred to as CSP **process templates**. *The equation $A_5 = A_6 \mid\mid\mid P_1$ denotes that a process executing A_5 transitions to launch two processes, one executing A_6 and the other executing P_1. The equation $P_3 = ?r \rightarrow P_5$ denotes that a process executing P_3 receives a message from process r and then transitions to P_5. The requester process executes template R. The definitions of other process template variables are similar. §3.1 contains a more comprehensive discussion of CSP.*

Second, the instrumenter takes as input a policy that specifies correct information flow. Policies are sets of declarative *flow assertions*, which are of two forms. A flow assertion of the form $\mathsf{Secrecy}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Declass}, \mathsf{Anc})$ specifies that any process executing template Source must not be able to send information to a process executing template Sink unless the information flows through a process executing template Declass, or the Source and Sink processes were transitively created by the *same process* executing template Anc. A flow assertion of the form $\mathsf{Prot}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Anc})$ specifies that if a process executing template Source attempts to send information to a process executing template Sink, and both the Source and Sink processes were transitively created by the same process executing template Anc, then the send must be successful.

Exa. 2. *The information-flow policy for the server in Fig. 1 can be expressed as the set of flow assertions:* $\{\mathsf{Secrecy}\ (W, W, \{P_1, P_2, P_3\}\ W), \mathsf{Prot}\ (W, P_2, R), \mathsf{Prot}\ (P_3, R, \mathsf{init})\}$*, where* init *is a special process template that spawns all process templates.*

In addition to flow assertions, the instrumenter takes a set of rules declaring which process templates denote processes that may be compromised. In Fig. 1, the instrumenter takes a rule declaring that the Worker may be compromised.

2.2 From Programs and Policies to Instrumentation Constraints

Given a program and policy, the instrumenter generates a system of set constraints such that a solution to the constraints corresponds to an instrumentation of the program that satisfies the policy. This constraint system must assert that (1) the instrumented program uses the Flume API to manipulate labels in a manner allowed by Flume, and (2) the instrumented program manipulates labels to satisfy all flow assertions.

Each variable in the constraint system corresponds to a label value for the set of all processes that execute a given CSP template. Flume restricts how a process manipulates its label in terms of the capabilities of the process. The instrumenter expresses this in the constraint system by generating, for each CSP process template P, variables that represent the set of tags in the label of a process executing P (lab_P), its positive capability (pos_P), its negative capability (neg_P), and the set of tags created when executing P ($\mathsf{creates}_P$).

Some constraints in the system assert that each process's labels may only change in ways allowed by Flume. These constraints assert that in each step of execution, a process's label may grow no larger than is allowed by its positive capability, and may shrink no smaller than is allowed by its negative capability.

Exa. 3. *To model how the label and capabilities of a process may change in transitioning from executing template A_1 to template A_5, the instrumenter generates the following four constraints:*

$$\mathsf{lab}_{A_5} \subseteq \mathsf{lab}_{A_1} \cup \mathsf{pos}_{A_5}$$
$$\mathsf{lab}_{A_5} \supseteq \mathsf{lab}_{A_1} - \mathsf{neg}_{A_5}$$
$$\mathsf{pos}_{A_5} \subseteq \mathsf{pos}_{A_1} \cup \mathsf{creates}_{A_5}$$
$$\mathsf{neg}_{A_5} \subseteq \mathsf{neg}_{A_1} \cup \mathsf{creates}_{A_5}$$

The other constraints in the system assert that the instrumented program does not allow flows that are specified by a Secrecy flow assertion, but allows all flows specfied in a Prot assertion.

Exa. 4. *For the server in Fig. 1, the flow assertion* $\mathsf{Secrecy}\ (W, W, \{P_1, P_3, P_5\}, W)$ *specifies that a* Worker *process, which executes template W, may not send information to a different* Worker *process unless the workers are created by the same* MPM *process that executed template A_1, or the information flows through a process that executes a* Proxy *template P_1, P_3, or P_5. In §3.3.2, we define a set Dist_{A_1} of process templates, which the instrumenter uses to encode the flow assertion as*

$$\mathsf{lab}_W \not\subseteq \left(\mathsf{lab}_W - \bigcup_{R \in \mathsf{Dist}_{A_1}} \mathsf{creates}_R \right) \cup \bigcup_{Q \notin \{P_1, P_3, P_5\}} \mathsf{neg}_Q$$

For illustrative purposes, this constraint is slightly simplified from its complete form, and may not lead the instrumenter to isolate Worker *processes that are compromised. However, the constraint can be extended to do so in a straightforward way using techniques described in App. C.*

Exa. 5. *For the server in Fig. 1, the flow assertion* $\mathsf{Prot}\ (W, P_3, A_1)$ *specifies that a* Worker *process executing template W must be able to send information to a process executing* Proxy *template P_3 if the two processes were created by the same* MPM *process executing template A_1. In §3.3.2, we define a set Const_{A_1} of process templates, which the instrumenter uses to encode the flow assertion as*

$$\mathsf{lab}_W \subseteq \mathsf{lab}_{P_3} \cap \bigcup_{Q \in \mathsf{Const}_{A_1}} \mathsf{creates}_Q$$

A similar flow assertion $\mathsf{Prot}(P_5, R, \mathsf{init})$ *specifies that a* Proxy *process executing P_5 must always be able to send data to the* Requester *executing R. The instrumenter encodes this assertion as*

$$\mathsf{lab}_{P_5} \subseteq \mathsf{lab}_R \cap \bigcup_{Q \in \mathsf{Const}_{\mathsf{init}}} \mathsf{creates}_Q$$

The constraints generated for the example from Fig. 1 are given in Tab. 1. The table contains a representative sample of the semantic constraints: analogous constraints are generated for the other process templates in the program.

2.3 From Instrumentation Constraints to DIFC Code

Solving the constraint systems described in §2.2 is an **NP**-complete problem. Intuitively, the complexity arises because such a constraint system may contain both positive and negative subset constraints and union set expressions, as shown in Tab. 1 See [8, App. G] for a formal proof. However, we can show that if the constraint system has a solution, then it has a solution in which all variables have a value with no more than N tags, where N is the number of Secrecy assertions in the policy. Using the bound N, the instrumenter translates the system of set constraints to a system of bit-vector constraints such that the set-constraint system has a solution if and only if the bit-vector system has a solution. Bit-vector constraints can be solved efficiently in practice by an off-the-shelf SMT solver [7]. The translator thus feeds the bit-vector system to such a solver; if the solver determines that no solution exists for

Semantics	Secrecy	Protected Flows
$\mathsf{lab}_{A_5} \subseteq \mathsf{lab}_{A_1} \cup \mathsf{pos}_{A_5}$ $\mathsf{lab}_{A_5} \supseteq \mathsf{lab}_{A_1} - \mathsf{neg}_{A_5}$ $\mathsf{pos}_{A_5} \subseteq \mathsf{pos}_{A_1} \cup \mathsf{creates}_{A_5}$ $\mathsf{neg}_{A_5} \subseteq \mathsf{neg}_{A_1} \cup \mathsf{creates}_{A_5}$ $\ldots$	$\mathsf{lab}_W \not\subseteq \left(\mathsf{lab}_W - \bigcup_{R \in \mathsf{Dist}_{A_1}} \mathsf{creates}_R\right) \cup \bigcup_{Q \notin \{P_1, P_3, P_5\}} \mathsf{neg}_Q$	$\mathsf{lab}_W \subseteq \mathsf{lab}_{P_3} \cap \bigcup_{Q \in \mathsf{Const}_{A_5}} \mathsf{creates}_Q$ $\mathsf{lab}_{P_5} \subseteq \mathsf{lab}_R \cap \bigcup_{Q \in \mathsf{Const}_{\mathsf{init}}} \mathsf{creates}_Q$

Table 1. A representative selection of constraints for the Apache server.

the bit-vector constraints, then it produces an *unsatisfiable core*, which is a minimal set of constraints that are unsatisfiable. The instrumenter determines the subprogram of the original program and subset of flow assertions that contributed the constraints in the infeasible core. It reports to the user that it may not be possible to instrument the program to satisfy the policy, and as a programming aid, provides the subprogram and policies that contributed the infeasible core. On the other hand, if the SMT solver finds a solution to the bit-vector constraints, then the instrumenter translates this to a solution for the system of set constraints.

Using the solution to the set-constraint system, the instrumenter injects into the original program DIFC code that defines the label values of all processes over any execution of the program to correspond to the values in the constraint solution. By construction, the resulting program satisfies the given information-flow policy.

Exa. 6. *One* Secrecy *flow assertion contributes to the constraint system summarized in Tab. 1. The instrumenter thus determines that if the system has a solution, then it has a solution that can be defined over a set of one element. Using this bound, the instrumenter translates the system to an equisatisfiable bit-vector system, and feeds the bit-vector system to an SMT solver. The solver determines that the bit-vector system has a solution corresponding to the following set-valued solution defined over the set of elements $\{\tau\}$:*

X	lab_X	pos_X	neg_X	$\mathsf{creates}_X$
A_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
A_5	$\emptyset$	$\{\tau\}$	$\{\tau\}$	$\{\tau\}$
A_6	$\emptyset$	$\{\tau\}$	$\{\tau\}$	$\emptyset$
A_7	$\emptyset$	$\{\tau\}$	$\{\tau\}$	$\emptyset$
P_1	$\{\tau\}$	$\{\tau\}$	$\{\tau\}$	$\emptyset$
P_3	$\{\tau\}$	$\{\tau\}$	$\{\tau\}$	$\emptyset$
P_5	$\emptyset$	$\{\tau\}$	$\{\tau\}$	$\emptyset$
W	$\{\tau\}$	$\emptyset$	$\emptyset$	$\emptyset$
R	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

The instrumenter uses the solution to generate the DIFC code highlighted in Fig. 1. In the solution, $\mathsf{creates}_{A_5} = \{\tau\}$*, so the instrumenter inserts at line* A_5 *a call to create a tag t. In the solution,* $\mathsf{lab}_{P_1} = \mathsf{pos}_{P_1} = \mathsf{neg}_{P_1} = \{\tau\}$*, so the instrumeter rewrites spawns of* Proxy *processes so that all* Proxy *processes are initialized with t in their label, positive capability, and negative capability. In the solution,* $\mathsf{lab}_{P_3} = \{\tau\}$ *while* $\mathsf{lab}_{P_5} = \emptyset$*, so the instrumenter inserts at* P_5 *code that clears t from the label of the process. The final result is the full code given in Fig. 1.*

3. DIFC Instrumentation

We now discuss the DIFC instrumenter in more detail. We first formally describe the programs and policies that the instrumenter takes as input, and then describe each of the steps it takes to instrument a program.

3.1 DIFC Programs

The instrumenter analyzes programs in a variation of CSP that we call $\mathsf{CSP}_{\mathsf{DIFC}}$. Imperative programs are translated automatically to $\mathsf{CSP}_{\mathsf{DIFC}}$ programs using a straightforward translation method spelled out in [8, App. I]. The syntax of $\mathsf{CSP}_{\mathsf{DIFC}}$ is given in Fig. 3. A $\mathsf{CSP}_{\mathsf{DIFC}}$ program $\overrightarrow{P}$ is a set of equations, each of which binds a *process template* to a process-template variable. Intuitively, a process template is the "code" that a process may execute. For convenience, we sometimes treat $\overrightarrow{P}$ as a function from template variables to the templates to which they are bound.

$$
\begin{array}{rl}
\mathsf{Prog} := & \mathsf{PROCVAR}_1 = \mathsf{Proc}_1 \ldots \mathsf{PROCVAR}_n = \mathsf{Proc}_n \\
\mathsf{Proc} := & \mathsf{SKIP} \\
 & |\mathsf{PROCVAR} \\
 & |\mathsf{EVENT} \rightarrow \mathsf{PROCVAR} \\
 & |\mathsf{PROCVAR}_1 \;\Box\; \mathsf{PROCVAR}_2 \\
 & |\mathsf{PROCVAR}_1 \;|||\; \mathsf{PROCVAR}_2 \\
\mathsf{EVENT} := & \mathsf{ChangeLabel}(LABEL, LABEL, LABEL) \\
 & |\;\mathsf{CREATE}_\tau \\
 & |?\;\mathsf{PROC\_ID} \\
 & |!\;\mathsf{PROC\_ID}
\end{array}
$$

Figure 3. $\mathsf{CSP}_{\mathsf{DIFC}}$: a fragment of CSP used to model the behavior of DIFC programs. Events in gray are not contained in programs provided by the user. They are only generated by the DIFC instrumenter.

The semantics of $\mathsf{CSP}_{\mathsf{DIFC}}$ follows that of standard CSP [2], but is extended to handle labels. The state of a $\mathsf{CSP}_{\mathsf{DIFC}}$ program is a set of processes. Processes are scheduled non-deterministically to execute their next step of execution. The program state binds each process to:

1. A process template, which defines the effect on the program state of executing the next step of the process.
2. A label, positive capability, and negative capability, which constrain how information flows to and from the process.
3. A namespace of tags, which constrain what tags the process may manipulate.

We give $\mathsf{CSP}_{\mathsf{DIFC}}$ a trace semantics, which associates to every $\mathsf{CSP}_{\mathsf{DIFC}}$ program $\overrightarrow{P}$ the set of traces of *events* that $\overrightarrow{P}$ may generate over its execution. Events consist of:

1. One process taking a step of execution.
2. One process spawning another process.
3. One process sending information to another.
4. One process receiving information from another.

Whenever a process p bound to template variable X takes a step of execution, p generates an event $\mathsf{STEP}(X)$. p then spawns a fresh process p', generates an event $\mathsf{SPAWNS}(\mathsf{p}, \mathsf{p}')$, sets the labels of p' to its own label values, sets the tag namespace of p' equal to its own, and halts. However, when $\overrightarrow{P}(X) = \mathsf{ChangeLabel}(L, M, N) \rightarrow \mathsf{PROCVAR}_1$ or $\overrightarrow{P}(X) = \mathsf{CREATE}_\tau \rightarrow \mathsf{PROCVAR}_1$, no events are generated in the trace. This allows us to state desired properties of an instrumentation naturally using equality over traces (see §3.2.2). Note that this definition of $\overrightarrow{P}$ is purely conceptual: programs produced by the instrumenter do not generate fresh processes at each step of execution.

When a process p takes a step of execution, it may have further effects on the program state and event trace. These effects are determined by the template to which p is bound. The effects are as follows, according to the form of the template:

- SKIP: p halts execution.
- $\mathsf{PROCVAR}$: p initializes a fresh process p' to execute the template $\overrightarrow{P}(\mathsf{PROCVAR})$.
- $\mathsf{PROCVAR}_1 \Box \mathsf{PROCVAR}_2$: p chooses non-deterministically to initialize p' to execute either template $\mathsf{PROCVAR}_1$ or template $\mathsf{PROCVAR}_2$.
- $\mathsf{PROCVAR}_1 \ ||| \ \mathsf{PROCVAR}_2$: p spawns a fresh process p', which it initializes to execute template $\mathsf{PROCVAR}_1$, and a second fresh process p'', which it initializes to execute template $\mathsf{PROCVAR}_2$.
- $\mathsf{CREATE}_\tau \rightarrow \mathsf{PROCVAR}_1$: p creates a new tag t, binds it to the *tag identifier* τ in the *tag namespace* of p, and adds t to both the positive and negative capabilities of p'. Tag t is never bound to another identifier, so at most one tag created at a given CREATE template can ever be bound in the namespace of a process. However, multiple tags created at a template can be bound in the namespaces of multiple processes.
- $\mathsf{ChangeLabel}(L, M, N) \rightarrow \mathsf{PROCVAR}_1$: L, M, and N are sets of tag identifiers. p initializes p' to execute $\mathsf{PROCVAR}_1$, and attempts to initialize the label, positive capability, and negative capability of p' to the tags bound in the namespace of p to the identifiers in L, M, and N, respectively. Each initialization is only allowed if it satisfies the conditions enforced by Flume: (1) the label of p' may be no larger (smaller) than the union (difference) of the label of p and the positive (negative) capability of p, and (2) the positive (negative) capability of p' may be no larger than the union of the positive (negative) capability of p and capabilities for all tags created at p'.
- $! \ q \rightarrow \mathsf{PROCVAR}_1$: p attempts to send information to process q. For simplicity, we assume that a process may attempt to send information to any process, and make a similar assumption for when p attempts to receive information. p generates an event $p!q$ only if it successfully sends information; that is, the label of p is contained in the label of q. Process p then initializes p' to execute template $\mathsf{PROCVAR}_1$.
- $? \ q \rightarrow \mathsf{PROCVAR}_1$: p attempts to receive information from q. p generates an event $p?q$ only if it successfully receives information; that is, the label of p contains the label of q. Process p then initializes p' to execute template $\mathsf{PROCVAR}_1$.

$\mathsf{Tr}(\overrightarrow{P})$ denotes the set of all traces of events that program $\overrightarrow{P}$ may generate. A formal definition of $\mathsf{Tr}(\overrightarrow{P})$ is given in [8, App. E].

3.2 DIFC Policies

Policies give a formal condition for when one program is a correct instrumentation of another.

3.2.1 Syntax of DIFC Policies

A DIFC policy $\mathcal{F} = (\mathcal{V}, \mathcal{S}, \mathcal{R})$ contains two sets, $\mathcal{S}$ and $\mathcal{R}$, of flow assertions defined over a set $\mathcal{V}$ of template variables. $\mathcal{S}$ is a set of *secrecy assertions*, each of the form $\mathsf{Secrecy}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Declass}, \mathsf{Anc})$, with $\mathsf{Source}, \mathsf{Sink}, \mathsf{Anc} \in \mathcal{V}$ and $\mathsf{Declass} \subseteq \mathcal{V}$. $\mathcal{R}$ is a set of *protection assertions*, each of the form $\mathsf{Prot}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Anc})$, with $\mathsf{Source}, \mathsf{Sink}, \mathsf{Anc} \in \mathcal{V}$.

3.2.2 Semantics of DIFC Policies

The semantics of a policy $\mathcal{F} = (\mathcal{V}, \mathcal{S}, \mathcal{R})$ is defined by a satisfaction relation $\overrightarrow{P}' \models (\overrightarrow{P}, \mathcal{F})$, which defines when a program $\overrightarrow{P}'$ is a correct instrumentation of $\overrightarrow{P}$ according to $\mathcal{F}$. Program $\overrightarrow{P}'$ must satisfy three *instrumentation conditions*: *secrecy* ($\overrightarrow{P}' \models_S \mathcal{S}$), *transparency* ($\overrightarrow{P}' \models_T (\overrightarrow{P}, \mathcal{R})$), and *containment* ($\overrightarrow{P}' \models_C (\overrightarrow{P}, \mathcal{R})$), which are defined below.

Secrecy. If no execution of $\overrightarrow{P}'$ leaks information from a source to a sink as defined by $\mathcal{S}$, then we say that $\overrightarrow{P}'$ satisfies the *secrecy instrumentation condition* induced by $\mathcal{S}$. To state this condition formally, we first define a set of formulas that describe properties of a trace of execution T. For process p and template P, let $p \in P$ denote that p executes P in its next step of execution. Let $\mathsf{spawned}_T(a, p)$ hold when process a spawns process p over the execution of trace T:

$$\mathsf{spawned}_T(a, p) \equiv \exists i. \ T[i] = \mathsf{SPAWN}(a, p)$$

Let $\mathsf{IsAnc}(a, p, T)$ hold when process a is an *ancestor* of p under the $\mathsf{spawned}_T$ relation:

$$\mathsf{IsAnc}(a, p, T) \equiv \mathsf{TC}(\mathsf{spawned}_T)(a, p)$$

where TC denotes the transitive closure operator. Let $\mathsf{ShareAnc}(p, q, \mathsf{Anc}, T)$ hold when processes p and q share an ancestor in Anc:

$$\mathsf{ShareAnc}(p, q, \mathsf{Anc}, T) \equiv \exists a \in \mathsf{Anc}. \quad \mathsf{IsAnc}(a, p, T) \land \mathsf{IsAnc}(a, q, T)$$

Finally, let $\mathsf{InfFlow}_{D,T}(p, q)$ hold when information is sent and received directly from process p to process q over the execution of trace T, where neither p or q execute a template in D:

$$\mathsf{InfFlow}_{D,T}(p, q) \equiv \exists i < j. \ ((T[i] = p!q \land T[j] = q?p) \lor \mathsf{spawned}_T(p, q)) \land \ p, q \notin D$$

$\overrightarrow{P}'$ satisfies the secrecy condition induced by $\mathsf{Secrecy}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Declass}, \mathsf{Anc}) \in \mathcal{S}$ if for every execution of $\overrightarrow{P}'$, a process $p \in \mathsf{Source}$ only sends information to a process $q \in \mathsf{Sink}$ with the information flow avoiding all processes in $\mathsf{Declass}$ if the endpoints p and q share an ancestor process $a \in \mathsf{Anc}$. Formally, for every trace $T \in \mathsf{Tr}(\overrightarrow{P}')$, and every $p \in \mathsf{Source}$ and $q \in \mathsf{Sink}$, the following must hold:

$$\mathsf{TC}(\mathsf{InfFlow}_{\mathsf{Declass},T})(p, q) \implies \mathsf{ShareAnc}(p, q, \mathsf{Anc}, T)$$

If the formula holds for every secrecy assertion in $\mathcal{S}$, then $\overrightarrow{P}'$ satisfies the secrecy instrumentation condition induced by $\mathcal{S}$, denoted by $\overrightarrow{P}' \models_S \mathcal{S}$.

Transparency over protected flows. If an execution of $\overrightarrow{P}$ performs only information flows that are described by the set $\mathcal{R}$, then this execution must be possible in $\overrightarrow{P}'$. We call this condition *transparency*. Formally, let $T \in \mathsf{Tr}(\overrightarrow{P})$ be such that $\mathsf{ProtTr}(T, \mathcal{R})$ holds, where

$$\begin{aligned}\mathsf{ProtTr}(T, \mathcal{R}) \equiv \forall p,q. \quad & \mathsf{InfFlow}_{\emptyset,T}(p, q) \implies \\ & \exists \mathsf{Prot}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Anc}) \in \mathcal{R}. \\ & p \in \mathsf{Source} \land q \in \mathsf{Sink} \\ & \land \mathsf{ShareAnc}(p, q, \mathsf{Anc})\end{aligned}$$

If for every such T, it is the case that $T \in \mathsf{Tr}(\overrightarrow{P}')$, then $\overrightarrow{P}'$ satisfies the transparency condition induced by P and $\mathcal{R}$, denoted by $\overrightarrow{P}' \models_T (P, \mathcal{R})$.

Trace containment for protected flows. Finally, an instrumented program $\overrightarrow{P}'$ should not exhibit any behaviors solely over flows protected by $\mathcal{R}$ that are not possible in the input program P. We call this condition *trace containment*. Formally, let $T \in \mathsf{Tr}(\overrightarrow{P}')$. If $\mathsf{ProtTr}(T, \mathcal{R})$ holds, then it must be the case that $T \in \mathsf{Tr}(\overrightarrow{P})$.

If this holds for every trace of $T \in \mathsf{Tr}(\overrightarrow{P}')$, then $\overrightarrow{P}'$ satisfies the containment condition induced by $\overrightarrow{P}$ and $\mathcal{R}$, denoted by $\overrightarrow{P}' \models_C (\overrightarrow{P}, \mathcal{R})$.

Formal Problem Statement. The goal of the DIFC instrumenter is thus to take as input a program $\overrightarrow{P}$, a DIFC policy $\mathcal{F} = (\mathcal{V}, \mathcal{S}, \mathcal{R})$, and produce a program $\overrightarrow{P}'$ such that $\overrightarrow{P}' \models_S \mathcal{S}$, $\overrightarrow{P}' \models_T (\overrightarrow{P}, \mathcal{R})$, and $\overrightarrow{P}' \models_C (\overrightarrow{P}, \mathcal{R})$. If $\overrightarrow{P}'$ satisfies all three conditions, then it is a correct instrumentation of $\overrightarrow{P}$ according to $\mathcal{F}$, denoted by $\overrightarrow{P}' \models (\overrightarrow{P}, \mathcal{F})$.

3.3 From Programs and Policies to Instrumentation Constraints

The DIFC instrumenter takes as input a program $\overrightarrow{P}$ and policy $\mathcal{F}$. To produce a program $\overrightarrow{P}'$ such that $\overrightarrow{P}' \models (\overrightarrow{P}, \mathcal{F})$, the instrumenter generates a system of set constraints such that a solution to the system corresponds to $\overrightarrow{P}'$. The constraints generated ensure two key properties of $\overrightarrow{P}$: (1) $\overrightarrow{P}'$ only manipulates labels in a manner allowed by the Flume reference monitor, and (2) the values of labels of all processes in all executions of $\overrightarrow{P}'$ ensure that $\mathcal{F}$ is satisfied.

3.3.1 Constraint Variables and Their Domain

The constraint system is defined over a set of variables, where each variable describes how a process should manipulate its label and capabilities when it executes a given template. One natural candidate for the domain of such variables is a finite set of atomic elements, where each element corresponds to a tag created by the program. However, if the DIFC instrumenter were to use such a domain, then it could not produce a program that may create an unbounded set of tags over its execution. The instrumenter thus could not handle many real-world programs and policies of interest, such as the example described in §2. The domain of the constraint variables is thus a finite set of atomic elements where each element corresponds to *a tag identifier bound at a template* CREATE_τ *in the instrumented program.*

For each $\mathsf{CSP}_{\mathsf{DIFC}}$ template variable X in $\overrightarrow{P}$, the instrumenter generates four constraint variables: lab_X, pos_X, neg_X, $\mathsf{creates}_X$. Let τ be a tag identifier. If in a constraint solution, $\tau \in \mathsf{creates}_X$, then in $\overrightarrow{P}'$, the template P bound to X is rewritten to $\mathsf{CREATE}_\tau \rightarrow P$. If $\tau \in \mathsf{lab}_X$, then the label of process $p \in X$ executing $\overrightarrow{P}'$ contains a tag bound to τ. The analogous connection holds for variable pos_X and the positive capability of p, and the variable neg_X and the negative capability of p.

Exa. 7. *The constraint variables used by the instrumenter are illustrated in Exa. 6. Consider the templates A_5 and W. The solution in Exa. 6 defines $\mathsf{creates}_{A_5} = \{\tau\}$. Thus the instrumenter rewrites template A_5 so that when a process executes A_5, it creates a tag and binds the tag to identifier τ. The solution defines $\mathsf{lab}_W = \{\tau\}$, $\mathsf{pos}_W = \mathsf{neg}_W = \emptyset$. Thus in the instrumented program, the label of each* Worker *process contains a tag bound to τ, but each* Worker *process cannot add or remove such a tag from its label.*

3.3.2 Generating Semantic Constraints

The instrumenter must generate a system of constraints such that any solution to the system results in DIFC code that performs actions allowed by Flume. To do so, the instrumenter contrains how a process's labels and capabilities may change over each step of its execution.

For each equation that defines the $\mathsf{CSP}_{\mathsf{DIFC}}$ program, the instrumenter generates the set of constraints $\mathsf{SemCtrs}$ defined as follows:

$$\begin{aligned}
\mathsf{SemCtrs}(X = \mathsf{SKIP}) &= \emptyset \\
\mathsf{SemCtrs}(X = Y) &= \mathsf{StepCtrs}(X, Y) \\
\mathsf{SemCtrs}(X = \mathsf{EVENT} \rightarrow Y) &= \mathsf{StepCtrs}(X, Y) \\
\mathsf{SemCtrs}(X = Y \,\square\, Z) &= \mathsf{StepCtrs}(X, Y) \\
&\quad \cup \mathsf{StepCtrs}(X, Z) \\
\mathsf{SemCtrs}(X = Y \,|||\, Z) &= \mathsf{StepCtrs}(X, Y) \\
&\quad \cup \mathsf{StepCtrs}(X, Z)
\end{aligned}$$

$\mathsf{SemCtrs}$ is defined by a function $\mathsf{StepCtrs}$, which takes as input two template variables X and Y. $\mathsf{StepCtrs}$ generates a set of constraints that encode the relationship between the labels of a process $p \in X$ and the labels of process $p' \in Y$ that p spawns in a step of execution. One set of constraints in $\mathsf{StepCtrs}$ encodes that if a tag is bound to an identifier τ and is in the label of p', then the tag must be in the label of p, or it must be in the positive capability of p'. Formally:

$$\forall \tau . \tau \in \mathsf{lab}_Y \implies \tau \in \mathsf{lab}_X \vee \tau \in \mathsf{pos}_Y$$

Equivalently:

$$\mathsf{lab}_Y \subseteq \mathsf{lab}_X \cup \mathsf{pos}_Y$$

Additionally, if a tag is bound to τ in the label of p and is not in the negative capability of p', then the tag must be in the label of p'. Formally:

$$\forall \tau . \tau \in \mathsf{lab}_X \wedge \tau \notin \mathsf{neg}_Y \implies \tau \in \mathsf{lab}_Y$$

Equivalently:

$$\mathsf{lab}_Y \supseteq \mathsf{lab}_X - \mathsf{neg}_Y$$

The other constraints in $\mathsf{StepCtrs}$ encode that the capabilities of p' may only grow by the capabilities of tags that p' creates. If p' has a positive (negative) capability for a tag bound to an identifier τ, then either p must have the positive (negative) capability for the tag, or the tag must be created and bound to τ at p'. Formally:

$$\begin{aligned}
\forall \tau . \tau \in \mathsf{pos}_Y &\implies \tau \in \mathsf{pos}_X \vee \tau \in \mathsf{creates}_Y \\
\wedge\ \tau \in \mathsf{neg}_Y &\implies \tau \in \mathsf{neg}_X \vee \tau \in \mathsf{creates}_Y
\end{aligned}$$

Equivalently:

$$\begin{aligned}
\mathsf{pos}_Y &\subseteq \mathsf{pos}_X \cup \mathsf{creates}_Y \\
\mathsf{neg}_Y &\subseteq \mathsf{neg}_X \cup \mathsf{creates}_Y
\end{aligned}$$

Finally, the instrumenter constrains that no tag identifier τ is bound at multiple templates. Formally:

$$\forall X, Y, \tau .\ X \neq Y \implies \tau \notin \mathsf{creates}_X \cap \mathsf{creates}_Y$$

Equivalently:

$$\bigwedge_{X \neq Y \in \mathsf{Vars}(\overrightarrow{P})} \mathsf{creates}_X \cap \mathsf{creates}_Y = \emptyset$$

The instrumenter conjoins these constraints with the constraints generated from applying $\mathsf{SemCtrs}$ to all equations in $\overrightarrow{P}$ to form a system of constraints φ_{Sem}. Any solution to φ_{Sem} corresponds to a program in which each process manipulates labels as allowed by Flume.

3.3.3 Generating Policy Constraints

The instrumenter must constrain that the instrumented program satisfies the instrumentation conditions of §3.2.2. To do so, the instrumenter generates constraints for each flow assertion in the policy.

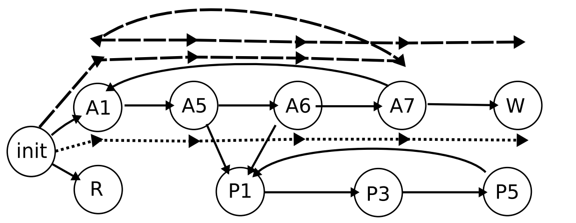

Figure 4. The spawn graph of the server from Fig. 1. The dotted and dashed paths denote process executions that invalidate init as a template to create tags that isolate Workers.

First, suppose that the instrumenter is given a secrecy assertion Secrecy(Source, Sink, Declass, Anc). The assertion induces a secrecy instrumentation condition. To instrument the program to respect this condition, the instrumenter must assert that, for processes $p \in \mathsf{Source}$ and $q \in \mathsf{Sink}$ that do not share an ancestor in Anc, information should not flow from p to q solely through processes that are not in Declass. We describe how the instrumenter asserts this by considering example executions that would violate the secrecy assertion. To describe these executions, we use the *spawn graph* of a program:

Def. 1. *For a* CSP$_{\mathsf{DIFC}}$ *program* $\overrightarrow{P}$*, the* **spawn graph** *of* $\overrightarrow{P}$ *is a graph that represents the "spawns" relation over process templates. Formally, the spawn graph of* $\overrightarrow{P}$ *is* $G_{\overrightarrow{P}} = (N, E)$*, where for every template variable* P*, node* $n_P \in N(G_{\overrightarrow{P}})$*, and* $(n_P, n_Q) \in E(G_{\overrightarrow{P}})$ *if and only if a process* $p \in P$ *may spawn process* $p' \in Q$*.*

The spawn graph of the program from Fig. 1 is given in Fig. 4.

Exa. 8. *Consider the server from Fig. 1 and the secrecy assertion that no* Worker*s executing* W *should be able to communicate information to each other unless the information flows through a proxy:* Secrecy$(W, W, \{P_1, P_2, P_3\}, W)$*. Suppose that the instrumenter generated no constraints to ensure that the instrumented program followed this assertion. The instrumenter might then instrument the program to create no tags. One execution of the program could then create a* Worker *process* p *by executing the series of templates* init*,* A_1*,* A_5*,* A_6*,* A_7*, and* W *(the dotted path in Fig. 4), and create another* Worker *process* q *by executing templates* init*,* A_1*,* A_5*,* A_6*,* A_7*,* A_1*,* A_5*,* A_6*,* A_7*, and* W *(the dashed path in Fig. 4). The label of* p *would then be a subset of the label of* q*, and thus* p *could send information to* q*.*

To guard against executions such as those in Exa. 8 for a general assertion Secrecy(Source, Sink, Declass, Anc), the instrumenter could generate the constraint $\mathsf{lab}_{\mathsf{Source}} \not\subseteq \mathsf{lab}_{\mathsf{Sink}}$. However, this constraint may not allow the instrumenter to find valid instrumentations of the program in important cases.

Exa. 9. *Suppose that for a general secrecy assertion* Secrecy(Source, Sink, Declass, Anc)*, the instrumenter generated the constraint* $\mathsf{lab}_{\mathsf{Source}} \not\subseteq \mathsf{lab}_{\mathsf{Sink}}$*. Then for the server of Fig. 1 and secrecy assertion of Exa. 8, the resulting constraint,* $\mathsf{lab}_W \not\subseteq \mathsf{lab}_W$*, is unsatisfiable, and the instrumenter would fail to instrument the server. However, if the instrumenter rewrote the server to create a tag each time a process executed template* A_1*, bind the tag to an identifier* τ*, and place the tag in the label of the next* Worker *spawned, then all* Worker *processes would be isolated.*

By contrast, if the instrumenter rewrote the server to create a tag each time a process executed template init*, bind the tag to an identifier* τ*, and place the tag in the label of the next* Worker *spawned, then there would be executions of the instrumented program in which* Worker *processes were not isolated. For example, the* Worker*s described in Exa. 8 would have in their labels the same tag bound to* τ*, and thus would be able to communicate.*

Thus there is a key distinction between templates A_1 *and* init*: if* p *and* q *are distinct* Worker *processes, then they cannot share the same tag created at* A_1*. However, they can share the same tag created at* init*.*

The instrumenter captures the distinction between A_1 and init in Exa. 9 for a general secrecy assertion Secrecy(Source, Sink, Declass, Anc) by constraining that there is a tag identifier $\tau \in \mathsf{lab}_{\mathsf{Source}}$ such that $\tau \notin \mathsf{lab}_{\mathsf{Sink}}$ *or* τ must be bound at a template in $\mathsf{Dist}_{\mathsf{Anc}}$, where $\mathsf{Dist}_{\mathsf{Anc}}$ is defined as follows:

Def. 2. *Let* P *and* Q *be process templates.* Q *is* **distinct** *for* P*, denoted by* $Q \in \mathsf{Dist}_P$*, if and only if the following holds. Let* Q *bind tags that it creates to a tag identifier* τ*, and let* r, s *be distinct processes with distinct ancestors in* P*. If* τ *is bound to a tag* t_1 *in the namespace of* r *and* τ *is bound to a tag* t_2 *in the namespace of* s*, then* $t_1 \neq t_2$*.*

To instrument a program to satisfy a secrecy assertion, the instrumenter could thus weaken the constraint $\mathsf{lab}_{\mathsf{Source}} \subseteq \mathsf{lab}_{\mathsf{Sink}}$ from above to $\mathsf{lab}_{\mathsf{Source}} \not\subseteq \mathsf{lab}_{\mathsf{Sink}} - \bigcup_{Q \in \mathsf{Dist}_{\mathsf{Anc}}} \mathsf{creates}_Q$. However, a program instrumented using such a constraint may still allow flows from a process $p \in \mathsf{Source}$ to a process $q \in \mathsf{Sink}$ not allowed by the assertion. The program could do so by allowing processes not in Declass to receive information from p, remove tags associated with the information, and then send the information to q. To guard against this, the instrumenter strengthens the above constraint to:

$$\mathsf{lab}_{\mathsf{Source}} \not\subseteq \left(\mathsf{lab}_{\mathsf{Sink}} - \bigcup_{Q \in \mathsf{Dist}_{\mathsf{Anc}}} \mathsf{creates}_Q \right) \cup \bigcup_{D \notin \mathsf{Declass}} \mathsf{neg}_D$$

We prove in [8, App. F] that this constraint is sufficient to ensure that the instrumented program satisfies the secrecy assertion.

Now suppose that the instrumenter is given a protection assertion Prot(Source, Sink, Anc). The assertion induces transparency and instrumentation conditions. To instrument the program to respect these conditions, the instrumenter must assert that whenever a process $p \in \mathsf{Source}$ communicates data to a process $q \in \mathsf{Sink}$ where p and q share an ancestor process $a \in \mathsf{Anc}$, then the communication must be successful. To assert this, the instrumenter must ensure that every tag t in the label of p is also in the label of q. We describe how the instrumenter does so by considering example executions that violate protection assertions.

Exa. 10. *Consider the server from Fig. 1 and the protection assertion* Prot(P_5, R, init) *that each* Proxy *executing* P_5 *must be able to send information to the* Requester *executing* R*. Suppose that the instrumenter generated no constraints to ensure that the program followed this assertion. The instrumenter might then instrument the program to bind a tag to an identifier* τ *at template* A_1*. One execution of the program could create a* Proxy *process* p *by executing the series of templates* init*,* A_1*,* A_5*,* A_6*,* P_1*,* P_3*, and* P_5*, and create a* Requester *process* q *by executing the series of templates* init *and* R*. Suppose that* p *had in its label the tag that was bound to* τ *when its ancestor executed* A_1*. Because no ancestor of* q *executed* A_1*,* q *would not have a tag in its label bound to* τ*. Thus the label of* p *would not be a subset of the label of* q*, and* p *would fail to communicate to* q*.*

Exa. 10 demonstrates that for assertion Prot(Source, Sink, Anc), if a tag in the label of $p \in \mathsf{Source}$ is bound to an identifier τ, then for p to send information to $q \in \mathsf{Sink}$, there must be a tag in the label of q that is bound to τ. This is expressed as $\mathsf{lab}_{\mathsf{Source}} \subseteq \mathsf{lab}_{\mathsf{Sink}}$. However, this constraint is not sufficient to ensure that p and q communicate, as demonstrated by the following example.

Exa. 11. *Suppose that the server in Fig. 1 was instrumented to bind a tag to an identifier τ at A_1, add this tag to the next* Proxy *process, and add the tag to the label of the* $\mathsf{Requester}$ *process executing R. Each* Proxy *process executing P_3 would have a different ancestor that executed A_1, and thus each* Proxy *would have a different tag in its label. Although $\mathsf{lab}_{P_3} = \{\tau\} \subseteq \{\tau\} = \mathsf{lab}_R$, because each tag bound to τ in the label of each* Proxy *process executing P_3 is distinct, the label of the* $\mathsf{Requester}$ *process does not contain the label of all processes executing P_3. Thus communication from* Proxy *processes to the* $\mathsf{Requester}$ *could fail.*

On the other hand, suppose that the server was instrumented to bind a tag to τ at init*, and add this tag to the label of* Proxy *processes executing P_3 and the* $\mathsf{Requester}$ *executing R. Then the same tag would be in the labels of each* Proxy *process and the* $\mathsf{Requester}$*. The key distinction between A_1 and* init *is that a* Proxy *and* $\mathsf{Requester}$ *may have distinct tags created at A_1, but cannot have distinct tags created at* init*.*

The instrumenter captures the distinction between init and A_1 in Exa. 11 for a general assertion $\mathsf{Prot}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Anc})$ by strengthening the above constraint that $\mathsf{lab}_{\mathsf{Source}} \subseteq \mathsf{lab}_{\mathsf{Sink}}$ to constraining that if a tag in the label of $p \in \mathsf{Source}$ is bound to an identifier τ, then $\tau \in \mathsf{lab}_{\mathsf{Sink}}$ *and* τ must be bound at a template in $\mathsf{Const}_{\mathsf{Anc}}$, where $\mathsf{Const}_{\mathsf{Anc}}$ is defined as follows.

Def. 3. *Let P, Q be process templates. Q is* **constant** *for P, denoted $Q \in \mathsf{Const}_P$, if and only if the following holds. Let processes r and s share in common their most recent ancestor in P, and let Q bind tags to a tag identifier τ. If τ is bound to a tag t_1 in the namespace of r, and τ is bound to a tag t_2 in the namespace of s, then $t_1 = t_2$.*

For a protection assertion $\mathsf{Prot}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Declass})$, the conditions on each tag identifier τ are expressed formally using $\mathsf{Const}_{\mathsf{Anc}}$ as:

$$\forall \tau . \tau \in \mathsf{lab}_{\mathsf{Source}} \implies \tau \in \mathsf{lab}_{\mathsf{Sink}} \land \tau \in \bigcup_{Q \in \mathsf{Const}_{\mathsf{Anc}}} \mathsf{creates}_Q$$

Equivalently:

$$\mathsf{lab}_{\mathsf{Source}} \subseteq \mathsf{lab}_{\mathsf{Sink}} \cap \bigcup_{Q \in \mathsf{Const}_{\mathsf{Anc}}} \mathsf{creates}_Q$$

We prove in [8, App. F] that this constraint is sufficient to ensure that the instrumented program satisfies the protection assertion.

The definitions of Dist and Const given in Defn. 2 and Defn. 3 explain how the sets are used to instrument a program, but they do not describe how the sets may be computed. The sets are computed through a series of reachability queries over the spawn graph of the program. For further details, see App. A.

A solution to the conjunction φ_{Pol} of constraints generated for all flow assertions in a policy $\mathcal{F}$ corresponds to an instrumentation that respects all assertions. A solution to the conjunction of these constraints with the semantic constraints, $\varphi_{\mathsf{Tot}} \equiv \varphi_{\mathsf{Sem}} \land \varphi_{\mathsf{Pol}}$, thus corresponds to a program that manipulates Flume labels legally to satisfy $\mathcal{F}$.

3.4 Solving Instrumentation Constraints

After generating a system of constraints φ_{Tot} as described in §3.3, the instrumenter must find a solution to φ_{Tot}, and from the solution instrument $\overrightarrow{P}$. Unfortunately, such systems are computationally difficult to solve in general; finding a solution to φ_{Tot} is $\mathbf{NP}$-complete in the number of terms in φ_{Tot}. We give a proof of hardness in [8, App. G].

However, although such systems are hard to solve in general, they can be solved efficiently in practice. Modern Satisfiability Modulo Theory (SMT) solvers [7] can typically solve large logical formulas very quickly. To apply an SMT solver, the instrumenter must translate φ_{Tot} from a formula over a theory of set constraints to a formula over a theory supported by the solver, such as the theory of bit-vectors. To translate φ_{Tot}, the instrumenter must derive for φ_{Tot} a bound B such that if φ_{Tot} has a solution, then it has a solution in which the value of each constraint variable contains at most B elements. Such a bound B always exists, and is equal to the number of secrecy flow assertions. We prove the validity of this bound and give the explicit rules for translating set constraints to bit-vector constraints in App. B.

The instrumenter applies an SMT solver to the bit-vector translation of the set-constraint system. If the SMT solver determines that the formula is unsatisfiable, then it produces an unsatisfiable core of bit-vector constraints. The core is a subset of the original constraint system that is unsatisfiable, and does not strictly contain an unsatisfiable subset. Given such a core, the instrumenter computes the subprogram and flow assertions that contributed constraints in the core, and presents these to the user. If the SMT solver determines that the constraint system is satisfiable, then the instrumenter rewrites the program so that the label values of all processes that execute the instrumented program correspond to the label values in the constraint solution.

3.5 From Constraint Solutions to Instrumented Programs

For a program $\overrightarrow{P}$ and policy $\mathcal{F}$, if the instrumenter obtains a solution to the constraint system φ_{Tot} described in §3.3.3, then from this solution it rewrites $\overrightarrow{P}$ to a new program $\overrightarrow{P}'$ that respects $\mathcal{F}$. Each equation $X = P$ in $\overrightarrow{P}$ is rewritten as follows. If $\mathsf{creates}_X$ contains a tag identifier τ, then the instrumenter rewrites P to $\mathsf{CREATE}_t \to P$. Now, suppose that L, M, and N are the sets of tag identifiers in the constraint values for lab_X, pos_X, and neg_X. Then the instrumenter rewrites P to $\mathsf{ChangeLabel}(L, M, N) \to P$. The instrumenter can reduce the number of $\mathsf{ChangeLabel}$ templates generated by only generating such a template when a label or positive capability changes from that of a preceeding P template in $G_{\overrightarrow{P}}$.

The instrumenter is sound in the sense that if it produces an instrumented program, then the program satisfies the instrumentation conditions of §3.2. However, it is not complete; e.g., to satisfy some programs and policies, it could be necessary for different processes executing the same template to contain tags created at different templates. This behavior is not supported by the instrumenter. However, our experiments, described in §4, indicate that in practice, the instrumenter can successfully instrument real-world programs to handle real-world policies.

4. Experiments

We evaluated the effectiveness of the DIFC instrumenter by experimenting with four programs. The experiments were designed to determine whether, for real-world programs and policies,

- the instrumenter is expressive: can its language of policies encode real-world information-flow policies, and can the instrumenter rewrite real programs to satisfy such policies? We found that each of the real-world policies could be encoded in the language of the instrumenter, and that the instrumenter could find a correct instrumentation of the program with minimal, if any, manual edits of the program.
- the instrumenter is efficient and scalable: can it instrument programs quickly enough to be used as a practical tool for developing applications? We found that the instrumenter could instrument programs in seconds.

To examine these properties, we implemented the DIFC instrumenter as an automatic tool [2] called SWIM [3] and applied it to instrument the following program modules:

1. The multi-process module of Apache [1].
2. The CGI and untrusted code launching modules of FlumeWiki [12].
3. The scanner module of ClamAV [4].
4. The OpenVPN client [16].

For each program module, we chose an information-flow policy from the literature [12, 21], expressed the policy in terms of the flow assertions described in §3.2.2, and then fed the program and policy to the tool.

We implemented SWIM using the CIL [15] program-analysis infrastructure for C, and the Yices SMT solver [7]. The only program annotations required by SWIM are C labels (*not* Flume labels) that map program points to variables used in flow assertions. When successful, SWIM outputs a C program instrumented with calls to the Flume API such that the program satisfies the input policy. We performed all experiments using SWIM on a machine with a 2.27 GHz 8-core processor and 6 GB of memory, although SWIM uses only a single core.

We first describe our experience using SWIM, and then evaluate its performance.

Apache Multi-Process Module. We applied SWIM to the multi-process module of the Apache web-server to automatically produce a version of Apache that isolates Worker processes. A model of the Apache system architecture, along with the desired policy, serves as the example described in §2. When we initially applied SWIM to Apache and its policy, SWIM was unable to find an instrumentation. Indeed, it was impossible to instrument Apache to isolate Worker processes, because its MPM process establishes a direct connection between the process that issues a request and the Worker process spawned to service a request. We thus modified the MPM code by hand to spawn *uninstrumented* proxy processes, which forward information between Worker and requester, as described in §1 and §2. When we applied the instrumenter to the modified program, it instrumented the new program so that it satisfied the policy. Although it is unfortunate that we had to modify the program manually, we were able to use the instrumenter to determine that such an modification was necessary. Moreover, the modification was relatively simple, and in the future, it may be possible to automatically rewrite a program as necessary with simple objects such as connection proxies to find a correct instrumentation.

FlumeWiki CGI and Untrusted Code Modules. We applied SWIM to FlumeWiki modules that launch processes to service requests, producing a version of FlumeWiki in which each process that services a request acts with exactly the DIFC permissions of the user who makes the request. FlumeWiki [12] is based on the software package MoinMoin Wiki [13], but has been extended to run on the Flume operating system with enhanced security guarantees. Similar to Apache, in FlumeWiki a *launcher* process receives requests from users for generating CGI forms, running potentially untrusted code, or interacting with the Wiki. The launcher then spawns an untrusted Worker to service the request. However, whereas Apache should execute with no information flowing from one Worker to another, in FlumeWiki each Worker should be able to access exactly the files that can be accessed by the user who issued the request. To express this policy and instrument FlumeWiki to satisfy it, we used policies defined over *persistent principals* (e.g. users). The semantics of these policies and the instrumenter's technique for generating code that satisfies them is analogous to how it generates code to handle the policies of §3.2. We give further details in [8, App. H]. We removed the existing DIFC code from the modules of FlumeWiki that launch processes that serve CGI forms or run untrusted code. We then applied SWIM to the uninstrumented program and policy. SWIM instrumented the program correctly, with code that was similar to the original, manually written code.

ClamAV Virus Scanner Module. We applied SWIM to ClamAV to automatically produce a virus scanner that is guaranteed not to leak sensitive data over a network or other output device, even if it is compromised. ClamAV is a virus-detection tool that periodically scans the files of a user, checking for the presence of viruses by comparing the files against a database of virus signatures. To function correctly, ClamAV must be trusted to read the sensitive files of a user, yet a user may want assurance that even if a process running ClamAV is compromised, it will not be able to send sensitive data over a network connection.

Inspired by [21], we modeled a system running ClamAV using the "scanner" module of ClamAV, a file containing sensitive user data, a file acting as a user TTY, a proxy between the scanner and the TTY, a file acting as a virus database, a file acting as a network connection, a process acting as a daemon that updates the virus database, and a process that spawns the scanner and update daemon and may set the labels of all processes and files. We then wrote a policy of nine flow assertions that specified that:

- The update daemon should always be able to read and write to the network and virus database.
- The scanner should always be able to read the sensitive user data and virus database.
- The scanner should never be able to send data directly to the network or TTY device. However, it should always be able to send data to the proxy, which should always be able to communicate with the TTY device.

SWIM automatically instrumented the model so that it satisfies the policy. Although we only used SWIM to instrument one, arbitrarily chosen system configuration, because SWIM is able to instrument systems very quickly, it could easily be used to reinstrument a system as the configuration of the system changes.

OpenVPN. We applied SWIM to OpenVPN to automatically produce a system that respects *VPN isolation.* OpenVPN is an open-source VPN client. Because VPNs act as a bridge between networks on both sides of a firewall, they represent a serious security risk [21]. A common desired policy for systems running a VPN client is VPN isolation, which specifies that information should be not able to flow from one side of a firewall to the other unless it passes through the VPN client.

We modeled a system running OpenVPN using the code of the entire OpenVPN program, files modeling networks on opposing sides of a firewall ($\mathsf{Network}_1$ and $\mathsf{Network}_2$), and a process (init) that launches OpenVPN and may alter the labels of the networks. We expressed VPN isolation for this model as a set of six flow assertions that specified that:

- Information show not flow between $\mathsf{Network}_1$ and $\mathsf{Network}_2$ unless it flows through OpenVPN.
- OpenVPN should always be able to read to and write from $\mathsf{Network}_1$ and $\mathsf{Network}_2$.

SWIM automatically instrumented the model so that it satisfies the policy. As in the case with ClamAV, we applied SWIM to one particular configuration of a system running OpenVPN, but SWIM is fast enough that it can easily be reapplied to a system running OpenVPN as the system's configuration changes.

Our experience using SWIM indicates that the DIFC instrumenter is sufficiently expressive to instrument real-world programs to satisfy real-world policies. While the flow assertions presented in §3.2 are simple to state, they can be combined to express complex,

[2] Available at http://cs.wisc.edu/∼wrharris/software/difc

[3] **S**ecure **W**hat **I** **M**ean

Program Name	LoC	Time (s)	Num. Inst.
Apache (MPM)	15,409	2.302	49
FlumeWiki (CGI)	300	0.183	46
FlumeWiki (WC)	286	0.096	34
ClamAV (scanner)	10,919	1.374	117
OpenVPN	98,262	7.912	51

Table 2. Performance of the DIFC instrumenter.

realistic policies. While not all programs could be instrumented to satisfy a desired policy without modification, when an instrumentation does exist, the instrumenter was able to find it each time.

For each application, we empirically measured the performance of SWIM. Results are given in Tab. 4. Col. "LoC" gives the number of lines of code in the program modules given to SWIM. Col. "Time (s)" gives the time in seconds required for the instrumenter to instrument the program. Col. "Num. Inst." gives the number of statements instrumented by SWIM. The results indicate the DIFC instrumenter is a practical technique: SWIM is able to instrument large, real-world program modules in seconds. Thus it is fast enough even to be integrated into the edit-compile-run cycle of the software-development work cycle.

5. Related Work

Multiple operating systems support DIFC, including Asbestos [19], HiStar [21], and Flume [12]. These systems all provide low-level mechanisms that allow an application programmer to implement an information-flow policy. Our instrumenter complements these systems. Although we have restricted the discussion to Flume, our instrumenter can be easily generalized to allow for program transformations to instrument programs for other DIFC operating systems. By running automatically-instrumented programs on top of a DIFC operating system, a user obtains greater assurance of the end-to-end information-flow security of their application. Note that the Asbestos system attempts to lessen the amount of DIFC code in applications by implicitly redefining label values to allow communication when the process is capable of performing the redefinition. This has multiple disadvantages. Krohn and Tromer [11] demonstrate that such systems may allow forbidden information flows. Furthermore, such implicit behaviors can make programs more difficult to reason about.

Harris et. al. [9] apply a model checker for safety properties of concurrent programs to determine if a fully instrumented DIFC application satisfies a high-level information flow policy. The present paper describes how to instrument DIFC code automatically, given only an uninstrumented program and a policy. Such code is correct by construction. Krohn and Tromer [11] use CSP to reason about the Flume OS, not applications running atop Flume.

Resin [20] is a language runtime that allows a programmer to specify dataflow assertions, which are checked over the state of the associated data before the data is allowed to be sent from one system object to another. Resin allows for arbitrary code to be run on certain events, but it does not attempt to provide guarantees that an application satisfies a high-level policy. In comparison, our policy language is less expressive, but the code generated by our approach is correct by construction. Additionally, DIFC systems provide certain guarantees that Resin does not match [20].

Previous work describes techniques to automatically synthesize programs from complete specifications of their behavior [6, 17]. Like our instrumenter, these techniques assume a program skeleton and a specification of correctness, and then use constraint solving to generate language constructs, yielding a concrete implementation of the specification. However, these techniques synthesize single-process arithmetic programs, while our instrumenter rewrites programs that may execute over an unbounded set of processes.

Several programming languages, such as Jif, provide type systems based on security labels that allow the programmer to validate security properties of their code through type-checking [14, 18]. Jif has been used to implement several real-world applications with strong security guarantees (e.g. [5, 3, 10]), but these programs are written from scratch in Jif. Automatic techniques can partition a Jif web application between its client and server [3], but the implementation of the composite program was done manually prior to partitioning. Our instrumenter adds labeling code to complete applications developed without label code.

6. Conclusion

Until now, the promise of DIFC operating systems has been limited by the added burden that they place on application programmers. We have presented a technique that takes a DIFC-unaware application and an information-flow policy and automatically instruments the application to satisfy the policy, while respecting the functionality of the application. Our technique thus greatly improves the applicability of DIFC systems and the end-to-end reliability of applications that run on such systems.

References

[1] Apache. `http://www.apache.org`.

[2] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[3] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *SOSP*, 2007.

[4] Clamav. `http://www.clamav.net`.

[5] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. *SP*, 2008.

[6] M. Colón. Schema-guided synthesis of imperative programs by constraint solving. In *LOPSTR*, 2004.

[7] B. Dutertre and L. de Moura. The Yices SMT solver. `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[8] W. R. Harris, S. Jha, and T. Reps. DIFC Programs by Automatic Instrumentation. http://cs.wisc.edu/∼wrharris/publications/tr-1673.pdf, 2010.

[9] W. R. Harris, N. A. Kidd, S. Chaki, S. Jha, and T. Reps. Verifying information flow control over unbounded processes. In *FM*, 2009.

[10] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *ACSAC*, 2006.

[11] M. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. In *SP*, 2009.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[13] MoinMoin. The MoinMoin wiki engine, Dec. 2006.

[14] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, 2002.

[16] Openvpn. `http://www.openvpn.net`.

[17] S. Srivastava, S. Gulwani, and J. S. Foster. From program verfication to program synthesis. In *POPL*, 2010.

[18] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *SP*, 2008.

[19] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[20] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, 2009.

[21] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

A. Computing Dist_P and Const_P

The definition of Dist given in Defn. 2 explains how Dist sets may be used to formulate constraints for $\mathsf{Secrecy}$ assertions, but it does not describe how Dist sets may be computed. For a program $\overrightarrow{P}$ with template P, the set Dist_P is computed by applying the following theorem and issuing a series of reachability queries over $\overrightarrow{P}$'s spawn graph $G_{\overrightarrow{P}}$:

Thm. 1. *For program $\overrightarrow{P}$ that contains process templates P and Q, $Q \in \mathsf{Dist}_P$ if and only if $\overrightarrow{P}$ contains no template variable R such that:*

1. n_Q reaches n_R.

2. $\overrightarrow{P}(R) = S \,|||\, T$, where n_S and n_T reach n_P without going through n_Q, or n_P reaches n_R and n_R reaches n_P without going through n_Q.

Proof. We first prove that if no such R exists, then $Q \in \mathsf{Dist}_P$. Let $T \in \mathsf{Tr}(\overrightarrow{P})$ be a trace of an arbitrary execution. If T has a subsequence $\mathsf{SPAWN}(p_0, p_1), \mathsf{SPAWN}(p_1, p_2), \ldots, \mathsf{SPAWN}(p_{n-1}, p_n)$, then we say that T has a spawn trace $p_0 \rightarrow p_1 \rightarrow \ldots \rightarrow p_n$. All processes that execute in T start from a root process template A. Consider processes d_1 and d_2 with spawn traces $a \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow p_1 \rightarrow \ldots \rightarrow \ldots d_1$ and $a \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow p_2 \rightarrow \ldots \rightarrow d_2$ with $a \in A$, $q \in Q$, and $p_1, p_2 \in P$ where d_1 and d_2 share the tag created at q in their namespaces. We want to show that $p_1 = p_2$.

Suppose on the contrary that $p_1 \neq p_2$. Because the spawn traces of d_1 and d_2 start from the same process a, there must be some processes r, s, t such that $r \rightarrow s$ is in the trace of d_1 while $r \rightarrow t$ is in the spawn trace of d_2, with $s \neq t$. Let r, s, t be the first processes for which this is the case. First, suppose that r spawns before both p_1 and p_2. Then the spawn traces of d_1 and d_2 have subtraces $q \rightarrow \ldots \rightarrow r \rightarrow s \rightarrow \ldots \rightarrow p_1 \rightarrow \ldots \rightarrow d_1$ and $q \rightarrow \ldots \rightarrow r \rightarrow t \rightarrow \ldots \rightarrow p_2 \rightarrow \ldots \rightarrow d_2$. By hypothesis, d_1 and d_2 share the same tag created at Q, so neither subtrace executes a process in Q other than q. However, this implies that r executes a template R such that $\overrightarrow{P}(R) = S \,|||\, T$, and n_S and n_T reach n_P without going through n_Q. Thus R satisfies the conditions of Thm. 1, which violates our hypothesis. Thus r cannot spawn before p_1 and p_2.

Now suppose that r spawns after p_1, but before p_2. Then the spawn traces of d_1 and d_2 have subtraces $q \rightarrow \ldots \rightarrow p_1 \rightarrow \ldots \rightarrow r \rightarrow \ldots \rightarrow d_1$ and $q \rightarrow \ldots \rightarrow r \rightarrow \ldots \rightarrow p_2 \rightarrow \ldots \rightarrow d_2$ with q the only process in Q, $p_1, p_2 \in P$, and $r \in R$. These subtraces imply that n_P reaches n_R and n_R reaches n_P without going through n_Q. Thus R satisfies the conditions of Thm. 1, and violates our hypothesis. r may not spawn before p_1 and after p_2, by a symmetric argument. Finally, if r spawned after p_1 and p_2, then it would be the case that $p_1 = p_2$, which violates our hypothesis. Thus no such r can exist, and it must be the case that $p_1 = p_2$.

We now prove that if R does exist, then $Q \notin \mathsf{Dist}_P$. First, suppose that $\overrightarrow{P}(R) = S \,|||\, T$, where n_S and n_T both reach n_P. Then $\overrightarrow{P}$ has an execution with processes d_1 and d_2 with spawn subtraces $q \rightarrow \ldots \rightarrow r \rightarrow s \rightarrow \ldots \rightarrow p_1 \rightarrow \ldots \rightarrow d_1$ and $q \rightarrow \ldots \rightarrow r \rightarrow t \rightarrow \ldots \rightarrow p_2 \rightarrow d_2$ that contain no processes in Q after q, with $r \in R$, $s \in S$, $t \in T$, and $p_1, p_2 \in P$. Thus d_1 and d_2 have distinct ancestor processes in P, but share a tag created at Q, and $Q \notin \mathsf{Dist}_P$ by definition. Now suppose that n_P reaches n_R, and n_S reaches n_P without going through n_Q. Then $\overrightarrow{P}$ has an execution with processes d_1 and d_2 with spawn subtraces $q \rightarrow \ldots \rightarrow p_1 \rightarrow \ldots \rightarrow r \rightarrow t \rightarrow d_1$ and $q \rightarrow \ldots \rightarrow p_1 \rightarrow r \rightarrow s \rightarrow \ldots \rightarrow p_2 \rightarrow d_2$ where $p_1 \neq p_2 \in P$, $r \in R$, $s \in S$, $t \in T$, and $s \neq t$. Neither subtrace contains a process in Q aside from the common q at the start of both traces, so d_1 and d_2 have distinct ancestor processes $p_1, p_2 \in P$, but share a tag created at Q. In the case that n_T instead of n_S reaches n_P, two processes d_1 and d_2 have distinct ancestor processes $p_1, p_2 \in P$ but share a tag created at Q, by a symmetric arguments. Thus by definition, $Q \notin \mathsf{Dist}_P$. □

The definition of Const given in Defn. 3 explains how Const sets may be used to formulate constraints for Prot assertions, but it does not describe how Const sets may be computed. For a program $\overrightarrow{P}$ with template P, the set Const_P is computed by applying the following theorem and issuing reachability queries over $\overrightarrow{P}$'s spawn graph $G_{\overrightarrow{P}}$:

Thm. 2. *For program $\overrightarrow{P}$ that contains process templates P and Q, $Q \in \mathsf{Const}_P$ if and only if $\overrightarrow{P}$ contains no template R such that:*

1. n_P reaches n_R in $G_{\overrightarrow{P}}$.

2. $\overrightarrow{P}(R) = S \,|||\, T$, where n_S or n_T reach n_Q without going through n_P.

Proof. We first prove that if there is no such R, then $Q \in \mathsf{Const}_P$. Let d_1, d_2 be distinct processes with spawn traces $a \rightarrow \ldots \rightarrow p \rightarrow \ldots \rightarrow r \rightarrow s \rightarrow \ldots \rightarrow d_1$ and $a \rightarrow \ldots p \rightarrow \ldots r \rightarrow t \rightarrow \ldots \rightarrow d_2$, where $a \in A$, $q \in Q$, $r \in R$, $s \in S$, $t \in T$, and $s \neq t$. By hypothesis, neither of the subtraces $s \rightarrow \ldots \rightarrow p_1$ nor $t \rightarrow \ldots \rightarrow p_2$ may execute Q, or R would satisfy the conditions of Thm. 2. Thus if d_1 has a tag created at Q, then it is the same tag that r has, which is the same tag that p_2 has.

We now prove that if such an R does exist, then $Q \notin \mathsf{Const}_P$. Suppose that n_S reaches n_Q, and the spawn traces from above have subtraces $s \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow d_1$ and $t \rightarrow \ldots \rightarrow d_2$, where $q \in Q$. When q executes Q, it creates a fresh tag u and binds u in its namespace. The only way that u may become unbound in the namespace is if a subsequent process executes Q again, binding a new fresh tag in the namespace. The most recent such tag is bound in the namespace of d_1. Thus d_1 cannot share a tag created by d_2, and $Q \notin \mathsf{Const}_P$. By a symmetric argument, if n_T reaches n_Q, then $Q \notin \mathsf{Const}_P$. □

B. Translating Set Constraints to Bit-vector Constraints

B.1 Bounding the Number of Abstract Tags in the Domain

To translate a system of set constraints φ to a system of bit-vector constraints, the instrumenter must derive a bound B such that if φ has a solution, then it has one in which the value of each variable has at most B elements. We will prove that B is equal to the number of negative set constraints in φ. To do so, we will prove a theorem over a general class of set constraints that include those that may be generated by the instrumenter.

Thm. 3. *Let φ be a conjunction of constraints, where each constraint is of the form $X \subseteq f(\overrightarrow{X})$ or $Y \not\subseteq g(\overrightarrow{Y})$, where X and Y are single variables, and $f(\overrightarrow{X})$ and $g(\overrightarrow{Y})$ are set expressions built* *only from the sets of variables $\overrightarrow{X}$ and $\overrightarrow{Y}$ using set union, intersection, and difference. If φ has a solution, then it has one in which the value for each variable contains at most one element for each negative subset constraint.*

Proof. Let s be a solution to φ, which maps each variable in φ to a set of atomic elements, and extends to set expressions in the natural way. For each negative subset constraint $Y \not\subseteq g(\overrightarrow{Y})$, pick one element $t \in s(Y) - s(g(\overrightarrow{Y}))$. Let E be the set of all such chosen elements, and for each variable X, let $s'(X) = s(X) \cap E$. s' is a solution to φ. To see this, first note that $s'(f(\overrightarrow{X})) = s(f(\overrightarrow{X})) \cap E$. This follows by the distributvity of intersection over union, intersection, and difference. To see that s' is a solution for any positive constraint in φ, observe that

$$s'(X) = s(X) \cap E \subseteq s(f(\overrightarrow{X})) \cap E = s'(f(\overrightarrow{X}))$$

To see that s' is a solution for any negative constraint in φ, observe that

$$s'(Y) = s(Y) \cap E \not\subseteq s(g(\overrightarrow{Y})) \cap E = s'(g(\overrightarrow{Y}))$$

The non-containment follows from the fact that by the definition of E, $s(Y)$ contains an element in E that is not in $s(g(\overrightarrow{Y}))$. □

B.2 From Set Constraints to Bit-vector Constraints

Using the bound B described in App. B.1, the instrumenter translates set-constraint system φ to a bit-vector constraint system φ_{bv} such that if φ_{bv} has a solution, then the solution can be translated efficiently into a solution for φ, and if φ_{bv} does not have a solution, then φ does not have a solution. To perform this translation, the instrumenter applies the following rules to each constraint in φ:

$$\begin{aligned}
[\![\mathsf{term}_1 \subseteq \mathsf{term}_2]\!]^B_C &\equiv [\![\mathsf{term}_1]\!]^B_T \& [\![\mathsf{term}_2]\!]^B_T = [\![\mathsf{term}_1]\!]^B_T \\
[\![\mathsf{term}_1 \not\subseteq \mathsf{term}_2]\!]^B_C &\equiv \neg [\![\mathsf{term}_1 \subseteq \mathsf{term}_2]\!]^B_C \\
[\![X]\!]^B_T &\equiv X^B_{bv} \\
[\![\mathsf{term}_1 \cup \mathsf{term}_2]\!]^B_T &\equiv [\![\mathsf{term}_1]\!]^B_T \mid [\![\mathsf{term}_2]\!]^B_T \\
[\![\mathsf{term}_1 \cap \mathsf{term}_2]\!]^B_T &\equiv [\![\mathsf{term}_1]\!]^B_T \& [\![\mathsf{term}_2]\!]^B_T \\
[\![\mathsf{term}_1 - \mathsf{term}_2]\!]^B_T &\equiv [\![\mathsf{term}_1]\!]^B_T \& \overline{[\![\mathsf{term}_2]\!]^B_T}
\end{aligned}$$

In the above rules, term is an arbitrary term in the theory of set constraints. $[\![\centerdot]\!]^B_C$ translates a set constraint to a constraint over bit-vectors of size B. It is defined using $[\![\centerdot]\!]^B_T$, which translates a set term to be a term defined over bit-vectors of size B. Variable X^B_{bv} denotes a bit-vector variable of size B that corresponds one-to-one with the set variable X, & denotes bit-wise AND, | denotes bit-wise OR, and $\overline{X}$ denotes the bit-wise complement of X.

C. Modeling Compromised Processes

A key strength of DIFC operating systems is that they allow a programmer to restrict the flow of their program's information, even if the information flows through a process that may be compromised. Obviously, the DIFC instrumenter cannot hope to instrument program segments that may become compromised and expect for the code to be run in the event of a compromise. However, the instrumenter can in many cases produce programs that soundly handle compromised processes.

To instrument code that may have compromised processes, we revise the constraints generated for policies. Prot assertions are given to ensure that the DIFC system preserves some aspect of the original program's functionality. It is hopeless to try to preserve functionality once a process has been compromised, so even if we know that processes mentioned in a Prot assertion may be compromised, the instrumenter does not rewrite the generated constraints.

However, suppose the instrumenter is given a secrecy assertion Secrecy(Source, Sink, Declass, Anc), and extra information that processes that execute Source or Sink may be compromised. The constraint generated for such an assertion as described in §3.3.3 is

$$\mathsf{lab}_{\mathsf{Source}} \not\subseteq \left(\mathsf{lab}_{\mathsf{Sink}} - \bigcup_{Q \in \mathsf{Dist}_{\mathsf{Anc}}} \mathsf{creates}_Q \right) \cup \bigcup_{D \notin \mathsf{Declass}} \mathsf{neg}_D$$

Suppose that processes executing Source may be compromised. If a compromised source process wishes to violate the assertion, then it will attempt to send information with the smallest possible label by removing all tags allowed by its negative capability. To constrain that such a process still cannot leak data, the instrumenter alters the left-hand side of the inequality:

$$(\mathsf{lab}_{\mathsf{Source}} - \mathsf{neg}_{\mathsf{Source}}) \not\subseteq \left(\mathsf{lab}_{\mathsf{Sink}} - \bigcup_{Q \in \mathsf{Dist}_{\mathsf{Anc}}} \mathsf{creates}_Q \right) \\ \cup \bigcup_{D \notin \mathsf{Declass}} \mathsf{neg}_D$$

Suppose that processes executing Sink may be compromised. If a compromised sink process wishes to violate the assertion, then it will attempt to receive information with the largest possible label by adding all tags allowed by its positive capability. To constrain that such a process still cannot leak data, the instrumenter alters the term in the right-hand side of the inequality related to the labels of Sink processes:

$$\mathsf{lab}_{\mathsf{Source}} \not\subseteq \left((\mathsf{lab}_{\mathsf{Sink}} \cup \mathsf{pos}_{\mathsf{Sink}}) \cup \bigcup_{Q \in \mathsf{Dist}_{\mathsf{Anc}}} \mathsf{creates}_Q \right) \\ \cup \bigcup_{D \notin \mathsf{Declass}} \mathsf{neg}_D$$

We can rewrite the constraints of §2.2 to obtain the full constraint system that allows the instrumenter to isolate Workers even if they are compromised.

D. Integrity

DIFC operating systems give application programmers the power to ensure the secrecy and *integrity* of their data. The techniques presented in the body of the paper automatically instrument programs to uphold secrecy guarantees. However, they can be extended to also instrument programs to uphold integrity guarantees. The full Flume operating system maps every process to an integrity label, in addition to the process's secrecy label, positive capability, and negative capability. To instrument programs to enforce integrity guarantees, the instrumenter generates for every template variable X an additional constraint variable int_X. The instrumenter generates constraints that assert the Flume semantics for integrity. These are directly analogous to those that assert the semantics of secrecy labels. The function $\mathsf{StepCtrs}(X, Y)$ defined in §3.3.2 is extended to generate the following constraints:

$$\mathsf{int}_Y \subseteq \mathsf{int}_X \cup \mathsf{pos}_Y$$
$$\mathsf{int}_Y \supseteq \mathsf{int}_X - \mathsf{neg}_Y$$

To extend the instrumenter to enforce integrity policies, we would define flow assertions for integrity. The instrumenter would then generate constraints analogous to those of §3.3.3.

E. Formal Trace Semantics of $\mathsf{CSP}_{\mathsf{DIFC}}$

We give a formal semantics of $\mathsf{CSP}_{\mathsf{DIFC}}$ programs as a denotational semantics that maps each program to the set of all event traces

$$
\begin{aligned}
\mathsf{ProcTr}((p, \mathsf{SKIP}), \pi, \lambda, \nu, \mu, \eta) =& \mathsf{STEP}(p, P).\mathsf{SPAWN}(p, p').\mathsf{ProgTr}(\pi - p, \lambda - p, \nu - p, \mu - p, \eta - p)\\
\mathsf{ProcTr}((p, X), \pi, \lambda, \nu, \mu, \eta) =& \mathsf{STEP}(p, P).\mathsf{SPAWN}(p, p').\\
& \mathsf{ProgTr}((\pi - p)[p' \mapsto \overrightarrow{P}(X)], \lambda[p' = p], \nu[p' = p], \mu[p' = p], \eta[p' = p])\\
\mathsf{ProcTr}((p, \mathsf{Proc}_1 \Box \mathsf{Proc}_2), \pi, \lambda, \nu, \mu, \eta) =& \mathsf{STEP}(p, P).\mathsf{SPAWN}(p, p').\\
& \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_1], \lambda[p' = p], \nu[p' = p], \mu[p' = p], \eta[p' = p]\\
& \cup \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_2], \lambda[p' = p], \nu[p' = p], \mu[p' = p], \eta[p' = p]))\\
\mathsf{ProcTr}((p, \mathsf{Proc}_1 \mathbin{|||} \mathsf{Proc}_2), \pi, \lambda, \nu, \mu, \eta) =& \mathsf{STEP}(p, P).\mathsf{SPAWN}(p, p').\mathsf{SPAWN}(p, p'') \;.\\
& \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_1, p'' \mapsto \mathsf{Proc}_2],\\
& \lambda[p' = p'' = p], \nu[p' = p'' = p], \mu[p' = p'' = p], \eta[p' = p])\\
\mathsf{ProcTr}((p, !\, q \to \mathsf{Proc}_1), \pi, \lambda, \nu, \mu, \eta) =& \mathsf{STEP}(p, P).\mathsf{SPAWN}(p, p').\ (\mathsf{if}(\lambda(p) \subseteq \lambda(q))\ \mathsf{then}\ (p!q)\ \mathsf{else}\ [\,]).\\
& \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_1], \lambda[p' = p], \nu[p' = p], \mu[p' = p], \eta[p' = p])\\
\mathsf{ProcTr}((p, ?\, q \to \mathsf{Proc}_1), \pi, \lambda, \nu, \mu, \eta) =& \mathsf{STEP}(p, P).\mathsf{SPAWN}(p, p').(\mathsf{if}(\lambda(q) \subseteq \lambda(p))\ \mathsf{then}\ (p?q)\ \mathsf{else}[\,]).\\
& \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_1], \lambda[p' = p], \nu[p' = p], \mu[p' = p], \eta[p' = p])\\
\mathsf{ProcTr}(p, \mathsf{ChangeLabel}(L, M, N) \to \mathsf{Proc}_1), \pi, \lambda, \nu, \mu, \eta) =& \mathsf{if}(\eta(p)(L) \subseteq \lambda(p) \cup \nu(p)) \wedge (\eta(p)(L) \supseteq \lambda(p) - \mu(p))\\
& \wedge (\eta(p)(M) \subseteq \nu(p)) \wedge (\eta(p)(N) \subseteq \mu(p))\\
& \mathsf{then}\ \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_1],\\
& \lambda[p' \mapsto L], \nu[p' \mapsto \eta(p)(M)], \mu[p' \mapsto \eta(p)(N)], \eta[p' = p])\\
& \mathsf{else}\ \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_1],\\
& \lambda[p' = p], \nu[p' = p], \mu[p' = p], \eta[p' = p])\\
\mathsf{ProcTr}((p, \mathsf{CREATE}_\tau \to \mathsf{Proc}_1), \lambda, \nu, \mu, \eta) =& \mathsf{ProgTr}((\pi - p)[p' \mapsto \mathsf{Proc}_1],\\
& \lambda[p' = p], (\nu - p)[p' \mapsto \nu(i) \cup \{t\}], (\mu - p)[p' \mapsto \nu(p) \cup \{t\}],\\
& (\eta - p)[p' \mapsto \eta(p)[\tau \mapsto t]), t \text{ fresh}\\
\mathsf{ProgTr}(\pi, \lambda, \nu, \mu, \eta) =& \bigcup_{(p \mapsto P) \in \pi} \mathsf{ProcTr}((p, P), \pi, \lambda, \nu, \mu, \eta)
\end{aligned}
$$

Figure 5. Trace semantics of $\mathsf{CSP}_{\mathsf{DIFC}}$.

that it may generate. Two functions define the trace semantics of a program $\overrightarrow{P}$. The function ProgTr defines the set of all traces that program $\overrightarrow{P}$ may generate from a given program state, and is defined as follows: first, let T be an infinite set of identifiers for tags created during execution of $\overrightarrow{P}$, and let $\mathcal{L} = 2^T$ be the set of all labels. Let I be the infinite set processes that may be spawned during execution. The function ProgTr takes the following partial functions as input:

1. A program state $\pi : I \rightharpoonup \mathsf{Proc}$, which maps each process to the template that it executes in its next step.
2. A label map $\lambda : I \rightharpoonup \mathcal{L}$, which maps each process to its current label.
3. A positive-capability map $\nu : I \rightharpoonup \mathcal{L}$, which maps each process to its positive capability.
4. A negative-capability map $\mu : I \rightharpoonup \mathcal{L}$, which maps each process to its negative capability.
5. A tag-namepsace map $\eta : I \rightharpoonup (A \rightharpoonup T)$, where A is the set of all tag identifiers in $\overrightarrow{P}$. η maps each process to a set of bindings from tag identifiers to tags. In other words, η maps each process to its namespace of tags.

Intuitively, ProgTr defines the set of traces generated by $\overrightarrow{P}$ from a state π to be the collection of traces generated from executing the next step of each process in π. By applying ProgTr to a program state that binds a single process to the root process template, we obtain the set of all traces that can be generated by the program. Formally, this set is expressed as $\mathsf{ProgTr}(\emptyset[p \mapsto R], \emptyset[p \mapsto \emptyset], \emptyset[p \mapsto \emptyset], \emptyset[p \mapsto \emptyset], \emptyset[p \mapsto \emptyset])$.

ProgTr is co-defined by a second function, ProcTr, which defines how executing one step of a process contributes to the execution trace of a program. Like ProgTr, the function ProcTr takes as input a program state, label map, positive-capability map, negative-capability map, and map to tag namespaces. In addition, ProcTr takes as its first argument a pair (p, P). Here, p is the identifier of the next process to take a step of execution, and P is the process template that it executes. When process p takes the next step of execution, it adds events to the execution trace and alters the program state based on the form of P. §3.1 informally describes what events are added and how the program state is altered. The equations in Fig. 5 define these effects formally.

The definitions in Fig. 5 make use of the following notation. Let f be an arbitrary map (in Fig. 5, f stands for either λ, ν, or μ). Then $f - i$ denotes f without a binding for domain element i. Map $f[i \mapsto d]$ is the same map as f, with the exception that domain element i is bound to range element d. Map $f[p' = p]$ is the same map as f, but with $f[p' = p](p') = f(p)$ and $f(p)$ undefined. Each equation has an implicit condition that any process spawned in the step of execution, such as p' or p'', is not previously in the domain of any map. The operator . is used to prefix a single event to a set of traces. If E is an event and $\mathcal{T}$ is a set of traces, then $E \,.\, \mathcal{T}$ denotes the set of traces $\{E \,.\, T \mid T \in \mathcal{T}\}$, where $E \,.\, T$ denotes prefixing the event E to the trace T.

F. Proof of Correctness

We now prove that when the instrumenter produces a program, the program is correct according to the definition of §3.2. We first give definitions and lemmas that allow us to prove this.

Def. 4. *Each trace of a program corresponds one-to-one with an* **execution** *of a program. An execution is a sequence of spawns $p \to q$, where p, q are processes, and $p \to q$ denotes that p executes the template to which it is bound, and then spawns q.*

Let $\sigma = p_0 \to p_1, \ldots, p_{n-1} \to p_n$ be an execution of program $\overrightarrow{P}$, and let $\overrightarrow{P}'$ be an instrumentation of $\overrightarrow{P}$ using the procedure given in §3. σ has a corresponding execution $\mathsf{inst}_{\overrightarrow{P}'}(\sigma)$ in $\overrightarrow{P}'$, which is the same with the exception of label manipulations, constructed as follows. Note that the instrumenter rewrites every equation $X = T$ in $\overrightarrow{P}$, to a set of equations of the form $X = E_0 \to X_0; X_0 = E_1 \to X_1; \ldots; X_n = T$, where each E_i is either a CREATE or $\mathsf{ChangeLabel}$ event. To construct $\mathsf{inst}_{\overrightarrow{P}'}$, rewrite σ by replacing each spawn $p \to q$ with $p \in T$ with a sequence of spawns $p \to p_0, p_1 \to p_2, \ldots, p_{n-1} \to p_n$, where each $p_i \in \overrightarrow{P}'(X_i)$. The resulting execution is in $\overrightarrow{P}'$.

Furthermore, for every execution in $\overrightarrow{P}'$, there is a corresponding execution $\mathsf{erase}(\sigma)$ in $\overrightarrow{P}$ that is the same, with the exception that it does not manipulate labels. Let $p \to q, q \to r$ be a pair of spawns in σ, an execution of $\overrightarrow{P}'$. If $q \in \overrightarrow{P}'(X)$ for some template X that is in $\overrightarrow{P}'$ but not in $\overrightarrow{P}$, then replace the pair of spawns with a single spawn $p \to r$. After repeating this for all such q, the resulting execution, $\mathsf{erase}(\sigma)$ is an execution of $\overrightarrow{P}$.

We now establish that if $\overrightarrow{P}'$ is instrumented to respect an protection assertion, then all flows described in the assertion are successful in all executions of $\overrightarrow{P}'$.

Lem. 1. *Let $\overrightarrow{P}$ be instrumented according to §3 for a policy $\mathcal{F} = (\mathcal{S}, \mathcal{R})$ to form $\overrightarrow{P}'$. Let σ be an execution of $\overrightarrow{P}'$ such if $p \to p'$ and $q \to q'$ are in σ where p executes template $!\ q \to X$ and q executes template $?\ p \to Y$, then there is some template* $\mathsf{Prot}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Anc}) \in \mathcal{R}$ *such that $p \in$* Source*, $q \in$* Sink*, and p and q share their most recent ancestor in* Anc*. Then $(p!q), (q?p) \in \mathsf{Tr}(\sigma)$.*

Proof. Let t be a tag where $t \in \mathsf{lab}_P$, and t is bound to an identifier τ. By the constraint of §3.3.3, $\mathsf{lab}_{\mathsf{Source}} \subseteq \mathsf{lab}_{\mathsf{Sink}}$, so τ is bound a tag u in the label of q. Furthermore, τ was bound at a template in $\mathsf{Const}_{\mathsf{Anc}}$. p and q share their most recent ancestor in Anc, so by the definition of $\mathsf{Const}_{\mathsf{Anc}}$, their tags bound to τ are the same. Thus $t = u$, and in general, the label of p is a subset of the label of q. Thus when p attempts to send data to q, the send is successful, and the program generates the event $p!q$. Symmetrically, when q attempts to receive data from p, it is successful, and the program generates the event $q?p$. □

An immediate corollary of Lem. 1 is that for an execution σ of $\overrightarrow{P}$, $\mathsf{Tr}(\sigma) = \mathsf{Tr}(\mathsf{inst}(\sigma))$. Furthermore, if σ is a trace of $\overrightarrow{P}'$, then $\mathsf{Tr}(\sigma) = \mathsf{Tr}(\mathsf{erase}(\sigma))$. We are now ready to prove that programs produced by the instrumenter are correct.

Thm. 4. *Given program $\overrightarrow{P}$ and policy $\mathcal{F} = (\mathcal{S}, \mathcal{R})$, let $\overrightarrow{P}'$ be the program produced by the instrumenter according to §3. Then $\overrightarrow{P}' \models (\overrightarrow{P}, \mathcal{F})$.*

Proof. To prove that $\overrightarrow{P}' \models (\overrightarrow{P}, \mathcal{F})$, we must prove that $\overrightarrow{P}'$ satisfies each of the instrumentation conditions given in §3.2.2. We first show that $\overrightarrow{P}'$ satisfies the secrecy instrumentation condition by considering an arbitrary secrecy flow assertion $\mathsf{Secrecy}(\mathsf{Source}, \mathsf{Sink}, \mathsf{Declass}, \mathsf{Anc}) \in \mathcal{S}$. Consider an arbitrary execution of $\overrightarrow{P}$ with a sequence of sends and receives $p \to p_1 \to \ldots \to p_n = q$, where $p \in \mathsf{Source}$, $q \in \mathsf{Sink}$, $p_i \notin \mathsf{Declass}$, and $p_i \to p_j$ denotes that p_i sends information to p_{i+1} and p_{i+1} later receives the information. We need only consider the case where p and q do not share an ancestor in Anc. By the constraints of §3.3.3 and the instrumentation of §3.5, the label of p contains a tag t created in some template P such that no tag created at P is in the negative capability of any process not in $\mathsf{Declass}$. Thus if a p_i received data from p_{i-1}, it had to have t in its label, and could not remove t from its label. By §3.3.3, either (1) no tag created at P is in the label of q or (2) P is distinct in Anc. If (1) holds, then the label of q does not contain t, and thus cannot contain the label of p_{n-1}. Thus p_{n-1} cannot send information to q. If (2) holds, then while the label of q may contain a tag u created at P, tags t and u must be distinct. Thus p_{n-1} still fails to send information to q. The secrecy flow assertion is thus respected, and in general $\overrightarrow{P}' \models_S \mathcal{S}$.

We now show that $\overrightarrow{P}'$ satisfies the transparency conditions induced by $\mathcal{R}$. Let $T \in \mathsf{Tr}(\overrightarrow{P})$ generated by execution σ. By Lem. 1, the trace of $\mathsf{inst}(\sigma)$ is identical, and thus $T \in \mathsf{Tr}(\overrightarrow{P}')$, and $\overrightarrow{P}' \models_T (\overrightarrow{P}, \mathcal{F})$. To see that $\overrightarrow{P}'$ satisfies the containment condition induced by, consider a trace $T \in \mathsf{Tr}(\overrightarrow{P}')$ generated by an execution σ for which all flows are protected by $\mathcal{R}$. By Lem. 1, $\mathsf{erase}(\sigma)$ has the same trace as σ. Thus $T \in \mathsf{Tr}(\overrightarrow{P})$, and $\overrightarrow{P}' \models_C (\overrightarrow{P}, \mathcal{F})$. □

G. Proof of Hardness of Constraint Solving

Let a *DIFC synthesis constraint problem* C be a constraint produced from the following grammar for Ctr:

$$\begin{aligned}
\mathsf{Ctr} &:= \mathsf{Ctr} \land \mathsf{Ctr} \\
&\mid \mathsf{VAR} \subseteq \mathsf{SetExpr} \\
&\mid \mathsf{VAR} \not\subseteq \mathsf{SetExpr} \\
\mathsf{SetExpr} &:= \emptyset \\
&\mid \mathsf{VAR} \\
&\mid \mathsf{SetExpr} \cup \mathsf{SetExpr} \\
&\mid \mathsf{SetExpr} - \mathsf{SetExpr}
\end{aligned}$$

Let V_C be the set of all variables appearing in C. A solution to C is some powerset lattice and a mapping from each variable in the constraint system to an element in the lattice that satisfies the constraints. Formally, it is a pair (n, s) where $n \in \mathbb{N}$ and $s : V_C \to 2^n$ is such that $s(c) = \mathbf{T}$, where $s(c)$ is the evaluation of variables lifted to an evaluation of expressions and constraints in the standard way. Let DIFC-SAT be the problem of deciding whether any DIFC synthesis constraint problem has a solution. Observe that while the grammar for Ctr does not describe all constraints that may be generated to instrument a DIFC program, for any constraint it is straightforward to construct a program and policy and induce the constraint.

Thm. 5. DIFC-SAT *is* $\mathbf{NP}$*-complete.*

Proof. We first show that the problem is in $\mathbf{NP}$. Observe that Thm. 3 applies to the constraints in DIFC-SAT. Thus by Thm. 3, if C has a solution, then it has one defined in a number of tags B that is linear in the number of constraints. For a non-determinisitc machine to solve an instance C in polynomial time, it guesses a solution (m, s) where m is bounded linearly by $|C|$. The assignment s can then be validated in a number of steps polynomial in the size of $|C|$.

We show completeness by reduction from 3SAT, first giving the reduction and then proving its correctness. Let φ be an instance of 3SAT. Assume without loss of generality that φ is in CNF. Generate a set variable $X_{\mathbf{T}}$ and constrain it with $X_{\mathbf{T}} \not\subseteq \emptyset$. For every propositional variable $x \in \mathsf{Vars}(\varphi)$, generate two set variables $X, \overline{X}$ and constrain them to be disjoint, expressed in the grammar of constraints as $X \subseteq X - \overline{X}$. Let $l_1 \vee l_2 \vee l_3$ be a disjunctive clause in φ. If $l_i = x_i$ is a variable, then let L_i be the corresponding set variable X_i, and if $l_i = \neg x_i$ is the negation of a variable, let L_i be $\overline{X_i}$. Construct the set constraint $X_{\mathbf{T}} \subseteq L_1 \cup L_2 \cup L_3$. Let $D(\varphi)$ be the collection of all constraints generated. $D(\varphi)$ has a DIFC solution if and only if φ has a solution.

We now prove that the given reduction is correct. First, suppose that the 3SAT instance φ has a satisfying assignment s. Then $D(\varphi)$ has a solution $(2^1, s')$ where $s'(X) = \{0\}$ if and only if $s(x) = \mathbf{T}$. To see this, observe that under s' each pair $X, \overline{X}$ is disjoint, trivially. Furthermore, it must be the case that $X_{\mathbf{T}} = \{0\}$, and each expression $X_{\mathbf{T}} \subseteq L_1 \cup L_2 \cup L_3$ corresponding to some clause $l_1 \vee l_2 \vee l_3$ must consist of at least one set variable that contains 0.

Now suppose that $D(\varphi)$ has a solution (n, s'). For a propositional literal x, let $x = \mathbf{T}$ if and only if $k \in s'(X_{\mathbf{T}})$ for any fixed $k < n$. For this to be a valid solution to φ, it must be the case that no propositional variable and its complement are both assigned to $\mathbf{T}$. However, this cannot be the case, as every pair of variables $X, \overline{X}$ is constrained to be non-overlapping, so it cannot be the case that both would contain any fixed element k. Additionally, it must be the case that for every $l_1 \vee l_2 \vee l_3$, in the corresponding set expression $L_1 \cup L_2 \cup L_3$, at least one set variable contains k. This follows because the set expression is constrained as $X_{\mathbf{T}} \subseteq L_1 \cup L_2 \cup L_3$, and $k \in X_{\mathbf{T}}$. □

H. Persistent Principals

Programs may sometimes need to be instrumented to have information flow to and from *persistent principals*, such as files, which may have labels and capabilities defined before the system is executed. To handle such cases, we extend the DIFC instrumenter beyond the techniques given in §3, which only reason about principals such as processes that are introduced during execution. Persistent principals are handled in an analogous way: the possibly unbounded set of labels that they may have are abstracted to a finite set tag identifiers, the requirements for how their information may flow are encoded as constraints over these identifiers, and the resulting constraint system is fed to a constraint solver. The key difference is that while in §3 each abstract tag represented all tags created when executing a template, with persistent principals each abstract tag represents all tags that a principal may have before execution.

Formally, we extend the set of flow assertions defined in §3.2 to include four additional assertions:

- $\mathsf{WriteAtMost}(\mathsf{user}, \mathsf{Proc})$ specifies that any process executing template Proc should only be able to write to files to which the persistent principal user is able to write.
- $\mathsf{WriteAtLeast}(\mathsf{user}, \mathsf{Proc})$ specifies that if a user is able to write to a file, then a process executing Proc should be able to write to it as well.
- $\mathsf{ReadAtMost}(\mathsf{user}, \mathsf{Proc})$ specifies that any process executing template Proc should only be able to read from files from which the persistent principal user is able to read.
- $\mathsf{ReadAtLeast}(\mathsf{user}, \mathsf{Proc})$ specifies that if a user is able to from a file, then a process executing Proc should be able read from it as well.

We extend the constraint system of §3.3 to reason about such flow assertions. For every template variable X in a $\mathsf{CSP}_{\mathsf{DIFC}}$ program $\overrightarrow{P}$, we generate additional constraints variables $\mathsf{PersLab}$, $\mathsf{PersPos}$, and $\mathsf{PersNeg}$, which are used to store the labels, positive capabilities, and negative capabilities associated with persistent principals. For each persistent principal u in the set U of all persistent principals mentioned in the set of flow assertions, we generate fresh abstract tags l_u, p_u, and n_u. The represent, respectively, the set of all tags in the label, positive capability, and negative capability of u when the programs executes.

The constraints generated over persistent constraint variables are analogous to those for non-persistent principals, with one key difference. Recall from §3.3.1 that for a template X, the variables lab_X, pos_X, and neg_X share one domain of elements, and that the value of capability variables pos_X and neg_X constrains how the values in the label variable lab_X may change. This is feasible for processes, because all abstract tags in the values of these variables correspond to tags created during execution of the program. However, when dealing with persistent principals, the instrumenter cannot assume any relationship between the tags in the label of a user u and those in its positive or negative capabilities. Thus the instrumenter must assume that once the tags in a user's label have been added to the label of a process, those tags cannot be removed. Formally, for every pair of template variables X, Y given to $\mathsf{StepCtrs}$ as defined in §3.3.2, the instrumenter generates the constraint $\mathsf{PersLab}_X \subseteq \mathsf{PersLab}_Y$. No semantic constraints restrict how a process may add or remove persistent capabilities, as we assume that a process may add the capabilities of any user, and may always drop capabilities.

The instrumenter generates constraints to encode each flow assertion that appears in a policy. It generates constraints per assertion as follows. In each case, let l_u, p_u, and n_u be the abstract tags allocated for the label, positive capability, and negative capability of user u.

- $\mathsf{WriteAtMost}(\mathsf{user}, P)$: the instrumenter must constrain that the label of P contains the label of user, and that the negative capabilities of P are contained by the negative capabilities of user. Formally:

$$\mathsf{PersLab}_P \supseteq \{l_u\}$$
$$\mathsf{PersPos}_P \subseteq \{p_u\}$$

- $\mathsf{WriteAtLeast}(\mathsf{user}, P)$: the instrumenter must constrain that the label of P is contained by the label of user, and that the negative capabilities of P contain the negative capabilities of the user. Formally:

$$\mathsf{PersLab}_P \subseteq \{l_u\}$$
$$\mathsf{PersNeg}_P \supseteq \{n_u\}$$

- $\mathsf{ReadAtMost}(\mathsf{user}, P)$: the instrumenter must constrain that the label of P is contained by the label of user, and that the positive capabilities of P are contained by the negative capabilities of user. Formally:

$$\mathsf{PersLab}_P \subseteq \{l_u\}$$
$$\mathsf{PersPos}_P \subseteq \{p_u\}$$

- $\mathsf{ReadAtLeast}(\mathsf{user}, P)$: the instrumenter must constrain that the label of P contains the label of user, and that the positive capabilities of P contain the positive capabilities of user. For-

$$
\begin{aligned}
\mathsf{Prog} &:= \mathsf{stmt} \\
\mathsf{stmt} &:= \texttt{VAR} := \mathsf{exp} \\
&\quad |\texttt{send}(p) \\
&\quad |\texttt{recv}(p) \\
&\quad |\texttt{spawn}(P) \\
&\quad |\mathsf{stmt}_1 ; \mathsf{stmt}_2 \\
&\quad |\texttt{if}\ \mathsf{exp}\ \texttt{then}\ \mathsf{stmt}_1\ \texttt{else}\ \mathsf{stmt}_2 \\
&\quad |\texttt{while}\ (\mathsf{exp})\ \mathsf{stmt}_1
\end{aligned}
$$

Figure 6. IMP: a simple imperative language.

mally:

$$
\begin{aligned}
\mathsf{PersLab}_P &\supseteq \{l_u\} \\
\mathsf{PersPos}_P &\supseteq \{p_u\}
\end{aligned}
$$

The instrumenter generates code based on a constraint solution as follows. The instrumenter represents a persistent principals as a quadruple of four objects:

1. A program variable that contains a *user token*.
2. A *label accessor function* that takes as input the user token and returns the persistent label of the persistent principal.
3. A *positive capability accessor function* that takes as input the user token and returns the positive capability of the persistent principal.
4. A *negative capability accessor function* that takes as input the user token and returns the negative capability of the persistent principal.

Flume provides such accessor functions for users in its API. Other accessors may be defined and given to the instrumenter as needed.

Persistent principals are used to instrument the FlumeWiki server described in §4. FlumeWiki requires that a Worker process execute with exactly the ability to write to and read from exactly the same files as the user who issues the request. This requirement is expressed as a conjunction of the flow assertions $\mathsf{WriteAtMost}(\mathsf{user}, \mathsf{Worker})$, $\mathsf{WriteAtLeast}(\mathsf{user}, \mathsf{Worker})$, $\mathsf{ReadAtMost}(\mathsf{user}, \mathsf{Worker})$, and $\mathsf{ReadAtLeast}(\mathsf{user}, \mathsf{Worker})$.

I. Instrumenting Imperative Programs

§3 considers how to instrument $\mathsf{CSP}_{\mathsf{DIFC}}$ programs. In practice, this is not a limitation, because imperative programs can be modeled as $\mathsf{CSP}_{\mathsf{DIFC}}$ programs, and an instrumented $\mathsf{CSP}_{\mathsf{DIFC}}$ program can be translated to an instrumented imperative program.

I.1 From Uninstrumented Imperative Programs to Uninstrumented CSP Programs

Imperative programs can be automatically modeled as $\mathsf{CSP}_{\mathsf{DIFC}}$ programs. To demonstrate this, we present a toy imperative language, IMP, whose syntax is given in Fig. 6. An IMP program is an imperative statement, which may optionally have a control-flow label. Statements can be constructed as normal assignments, the distinguished statements `send`, `recv`, and `spawn`, and the standard control-flow structures if and while. Calls to `send` and `recv` communicate information to a process identified as p, while $\texttt{spawn}(Q)$ launches a new process starting execution at the statement labeled Q, while continuing execution of the current process. The set of expressions that can occur in an IMP program is ignored by the instrumenter, and thus we omit a detailed description. The semantics of IMP programs are standard.

An IMP program P can be translated into a program in $\mathsf{CSP}_{\mathsf{DIFC}}$ that over-approximates the set of executions of P. Such a translation is described by the rules in Fig. 7, which are written in an ML-like syntax. In Fig. 7, the function $\mathsf{ImpToCSP}$ takes as input an IMP program to be translated and an auxiliary $\mathsf{CSP}_{\mathsf{DIFC}}$ process variable P. Variable P is assumed to be bound by a process equation to a $\mathsf{CSP}_{\mathsf{DIFC}}$ process that models all execution after the IMP program. The output of $\mathsf{ImpToCSP}$ is a list of equations defining all process templates. Each rule that partially defines $\mathsf{ImpToCSP}$ introduces a fresh process variable X. The variable X binds to the $\mathsf{CSP}_{\mathsf{DIFC}}$ process representation of the IMP statement being translated.

The translation $\mathsf{ImpToCSP}$ yields a CSP program whose trace semantics "reasonably" *over*-approximates the set of traces of the original IMP program in the following sense. If one were to replace all predicates in all guards in the original IMP program with a predicate whose value was non-deterministic, then the set of traces of the IMP program and $\mathsf{CSP}_{\mathsf{DIFC}}$ process obtained by translation would be identical.

I.2 Instrumented $\mathsf{CSP}_{\mathsf{DIFC}}$ Programs to Instrumented Imperative Programs

If we translate imperative programs to $\mathsf{CSP}_{\mathsf{DIFC}}$ programs using the rules of Fig. 7, then every imperative program statement corresponds to exactly one $\mathsf{CSP}_{\mathsf{DIFC}}$ template variable. Suppose that the instrumenter produces an instrumented $\mathsf{CSP}_{\mathsf{DIFC}}$ program that satisfies a policy. The instrumented program contains templates of two new forms that must be translated to a corresponding form in the imperative program. First, a $\mathsf{CSP}_{\mathsf{DIFC}}$ program may contain a template of the form $\mathsf{CREATE}_t \rightarrow P$. To translate such a template to an imperative program, the instrumenter declares in the imperative program a global tag variable `tag_t` and instruments the corresponding location with a call `t := create_tag()`.

Second, a $\mathsf{CSP}_{\mathsf{DIFC}}$ program may contain a template of the form $\mathsf{ChangeLabel}(L, M, N) \rightarrow P$, where L, M, N are sets of tag identifiers. The translator prepends the corresponding program location with calls to the Flume API function that set the label values of the process to the corresponding label values. Let $t \in L$ be a tag identifier. The instrumenter generates code that uses an empty scratch variable (e.g. `tmp`), and adds to it the tag stored in the corresponding label variable `tag_t`.

Exa. 12. *Suppose that* `x := exp` *is a statement in an imperative program given to the DIFC instrumenter, and that the instrumenter translates the statement to a* $\mathsf{CSP}_{\mathsf{DIFC}}$ *program in which the statement is modeled by an equation* $X = P$*. Suppose that the instrumenter finds a solution for the program containing the statement and a policy, and rewrites the* $\mathsf{CSP}_{\mathsf{DIFC}}$ *equation to be* $X = \mathsf{ChangeLabel}(L, M, N) \rightarrow P$*, where* $L = \{t\}$*,* $M = \{t, u\}$*, and* $N = \emptyset$*. The instrumenter then generates the following imperative code:*

```
clear_tag_set(tmp);
expand_tag_set(tmp, tag_t);
set_label(tmp);
clear_tag_set(tmp);
expand_tag_set(tmp, tag_t);
expand_tag_set(tmp, tag_u);
set_pos_cap(tmp);
clear_tag_set(tmp);
set_neg_cap(tmp);
x := exp;
```

$$
\begin{aligned}
\mathsf{ImpToCSP}(x := \mathsf{exp}, P) &= [\mathsf{STEP}(X) \rightarrow P] \\
\mathsf{ImpToCSP}(\texttt{send}(c), P) &= [X =!\; c \rightarrow P] \\
\mathsf{ImpToCSP}(\texttt{recv}(c), P) &= [X =?\; c \rightarrow P] \\
\mathsf{ImpToCSP}(\texttt{spawn}(Q), P) &= [X = P \;|||\; Q] \\
\mathsf{ImpToCSP}(\mathsf{stmt}_1; \mathsf{stmt}_2, P) &= \\
\mathsf{let}[Y_1 = Z_1; \ldots]\ \mathsf{as\ defs} &= \mathsf{ImpToCSP}(\mathsf{stmt}_2, P)\ \mathsf{in} \\
&\quad [X = \mathsf{ImpToCSP}(\mathsf{stmt}_1, Y_1)]\ .\ \mathsf{defs} \\
\mathsf{ImpToCSP}(\texttt{if}\ \mathsf{exp}\ \texttt{then}\ \mathsf{stmt}_1\ \texttt{else}\ \mathsf{stmt}_2, P) &= \\
\mathsf{let}[Y_1 = Z_1; \ldots]\ \mathsf{as\ defs}_1 &= \mathsf{ImpToCSP}(\mathsf{stmt}_1, P)\ \mathsf{in} \\
\mathsf{let}[Y_2 = Z_2; \ldots]\ \mathsf{as\ defs}_2 &= \mathsf{ImpToCSP}(\mathsf{stmt}_2, P)\ \mathsf{in} \\
[X = Y_1 \Box\, Y_2].\mathsf{defs}_1.\mathsf{defs}_2 & \\
\mathsf{ImpToCSP}(\mathsf{while\ (exp)\ stmt}, P) &= \\
\mathsf{let}[Y_1 = Z_1; \ldots]\ \mathsf{as\ defs}_1 &= \mathsf{ImpToCSP}(\mathsf{stmt}, P)\ \mathsf{in} \\
&\quad [X = P \Box Y_1]\ .\ \mathsf{defs}_1
\end{aligned}
$$

Figure 7. Translation rules from IMP statements to $\mathsf{CSP}_{\mathsf{DIFC}}$ processes.