DIFC Programs by Automatic Instrumentation *

William R. Harris
Univ. of Wisconsin, Madison
Dept. of Computer Sciences
wrharris@cs.wisc.edu

Somesh Jha
Univ. of Wisconsin, Madison
Dept. of Computer Sciences
jha@cs.wisc.edu

Thomas Reps †
Univ. of Wisconsin, Madison
Dept. of Computer Sciences
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 their 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 these tasks. 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 implemented it in an automatic tool, called SWIM (for **S**ecure **W**hat **I** **M**ean), and applied it to a set of real-world programs and policies. The results of our evaluation demonstrate that the technique is sufficiently expressive to produce programs for real-world policies, and that it can produce such programs efficiently. It thus represents a significant contribution towards developing systems with strong end-to-end information-flow guarantees.

Categories and Subject Descriptors

D.2.4 [**Software / Program Verification**]: Formal methods; D.4.6 [**Security and Protection**]: Information flow controls

General Terms

Languages, Security

Keywords

Constraint Solving, DIFC, Instrumentation

1. INTRODUCTION

Decentralized information flow control (DIFC) operating systems are a recent innovation aimed at providing applications with mechanisms for ensuring the secrecy and integrity of their data [14, 21, 23]. To achieve this goal, a DIFC OS associates with each process a label drawn from a partially-ordered set. A process may send data to another process only if the processes' labels satisfy a certain ordering, i.e., $label(sender) \subseteq label(receiver)$. A process may change its own labels, subject to certain restrictions enforced by the DIFC OS. Thus, processes express how they intend their information to be shared by attaching labels to OS objects, 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 [13]. 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 [14]. 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 potentially wide applicability. The work in [14] gives a comprehensive description of the Flume system.

†Also affiliated with GrammaTech, Inc.

*Supported by NSF under grants CCF-0540955, CCF-0810053, CCF-0904371, and CNS-0904831, by ONR under grant N00014-09-1-0510, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, ONR, ARL, and AFRL.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

```
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. Key: `Typewriter typeface`: original code; `underlined code`: code for proxy processes to mediate interprocess communication, added manually; `shaded code`: label-manipulation code, added automatically by SWIM.

Flume's label-manipulation primitives provide a low-level "assembly language" for enforcing security policies. To illustrate the gap between low-level label manipulations and high-level policies, consider the problem of isolating Worker processes in the server program shown in Fig. 1. Fig. 1 gives a simplified excerpt from an Apache multi-process module (MPM) [1]. Suppose that the program consists of all the non-shaded code (in particular, it includes the underlined 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 sent over the connection (line A7). Along with the Worker, the MPM process spawns two processes to serve as proxies for when the Worker sends information to the Requester (line A5) and receives information from the Requester (line A6).

Now consider the high-level policy that the Worker processes should be isolated from each other: no Worker process should be able leak information to another Worker process through a storage channel,[1] 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 multitude 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 shaded code, which makes use of Flume label-manipulation operations. The label manipulations ensure process isolation as follows. As before, an MPM process spawns a Worker process and two Proxy processes for each request. In addition, at line A4, the MPM process creates a fresh, atomic *tag*, and then initializes the label of the next Worker process to contain the tag, but does not give the Worker the capability to remove the tag from its label (line A7). This restriction on removing the tag is the key to isolating each Worker process. It causes the label of each Worker to contain a tag that the Worker cannot remove. Because 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, no Worker can send data successfully to another Worker. 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 has no control over 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. Consequently, 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. The MPM resolves this as follows. Let Proxy_S be the process spawned in line A5 to forward 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. Overall, the untrusted Worker is isolated, while the small and trusted Proxys can mediate the communication of information from the Worker to the 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. Currently, if programmers are to develop applications for DIFC systems, they must resolve these issues manually. If they wish to instrument existing applications to run on DIFC systems, 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 [14]. 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, called SWIM (for **S**ecure **W**hat **I** **M**ean), 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.").

[1]DIFC systems do not address timing channels.

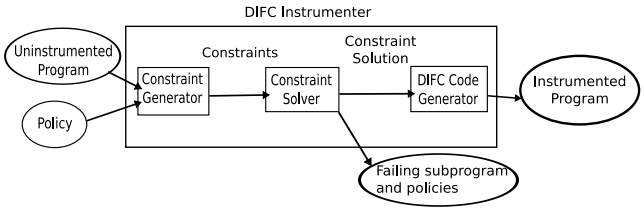

Figure 2: Workflow of the DIFC instrumenter, SWIM**.**

When successful, SWIM rewrites the input program with label-manipulation code so that it enforces the input policy. When unsuccessful, SWIM produces a minimal subprogram of the original program and a minimal subset of policies in conflict for which SWIM could not find an instrumentation. Thus, if a programmer provides policies that are in conflict, SWIM identifies a small subset of policies that are in conflict. To do so, SWIM 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 [6], which in our experiments found solutions to the systems in seconds (see Tab. 4). From a solution, SWIM instruments the program. Thus the programmer reasons about information flow at the policy level, and leaves to SWIM the task of correctly manipulating labels. If the programmer provides as input to SWIM the program in Fig. 1 without the shaded code, and a formal statement of a high-level policy similar to the one stated above, SWIM produces the entire program given in Fig. 1.

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 appendices of [9].

2. OVERVIEW

We now informally describe each step of the workflow of SWIM, using the example from Fig. 1. We first give a brief overview of the Flume operating system. For a more complete description, see [14].

Overview of Flume's DIFC Primitives. Flume is implemented as a reference monitor that runs as a user-level process, but uses a Linux Security Module for system-call interposition. Flume monitors Linux programs that have been modified to make use of Flume's DIFC primitives, which are described below.

For each process, Flume maintains a secrecy label, an integrity label, and a capability set. (SWIM currently only supports secrecy labels and capability sets.)

- **Tags and Labels**. A *tag* is an atomic element created by the monitor at the request of a process. A *label* is a set of tags associated with an OS object.
- **Capabilities**. An OS object can only alter its label by adding a tag t in its capability set marked as a *positive capability* (t^+), or by removing a tag t in its capability set marked as a *negative capability* (t^-).
- **Channels**. Processes are not allowed to create their own file descriptors. Instead, a process asks Flume for a new *channel*, and receives back a pair of *endpoints*. Endpoints may be passed to other processes, but each endpoint may be claimed by at most *one* process, after which they are used like ordinary file descriptors.

The monitor forbids a process p with label l_p to send data through endpoint e with label l_e unless $l_p \subseteq l_e$. Likewise, the monitor forbids data from endpoint e' to be received by a process p' unless $l_{e'} \subseteq l_{p'}$. A Flume process may create another process by invoking the `spawn` command. `spawn` takes as input (i) the path to the executable to be executed, (ii) an ordered set of endpoints that the new process may access from the beginning of execution, (iii) an initial label, and (iv) an initial capability set, which must be a subset of the capability set of the spawning process.[2]

We now describe the approach employed by SWIM by discussing each step of how SWIM instruments the example from Fig. 1. SWIM's workflow is depicted in Fig. 2.

Programs and Policies as Inputs. SWIM takes two inputs: a C program and a policy. The C program does not manipulate Flume labels. For the example in Fig. 1, SWIM receives the unshaded code (i.e., the version of the server without the calls to `create_tag`(), `expand_label()`, and `clear_label()`). SWIM represents programs internally using a dialect of Communicating Sequential Processes (CSP) [2]: it translates the C program into a CSP program that models properties relevant to DIFC instrumentation, and then analyzes the CSP program. We use CSP as an intermediate representation because CSP is a natural formalism for representing programs that involve an a priori unbounded set of processes.

EXA. 1. *The following are a representative sample of the equations generated when* SWIM *translates the unshaded code from Fig. 1 to CSP:*

$$A_5 = A_6 \,|||\, P_1 \qquad A_6 = A_7 \,|||\, P_1 \qquad A_7 = A_1 \,|||\, W$$
$$P_1 = P_3 \qquad P_3 = ?r \rightarrow P_5 \qquad P_5 = !s \rightarrow P_1$$
$$\mathsf{init} = A_1 \,|||\, R$$

In these equations, each variable corresponds to a program point in the C 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 $A_5 = A_6 \,|||\, P_1$ denotes that a process executing A_5 can make a transition that launches 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 can receive a message from endpoint r and then make a transition to P_5. The definitions of other process-template variables are similar.*

The process-template variables R and W refer to Requester *and* Worker *processes, respectively. They are defined by other CSP equations, which are not shown above.*

SWIM's second input is a policy that specifies the desired information flows. Policies are sets of declarative *flow assertions*, which are of two forms. A flow assertion of the form Secrecy(Source, Sink, Declass, Anc) specifies that any process executing template Source must not be able to send information to a process executing template Sink unless (i) the information flows through a process executing template Declass,

[2]In our examples, we use two sets of explicit positive and negative capabilities, rather than one set with capabilities marked with $^+$ or $^-$.

or (ii) the Source and Sink processes were transitively created by the *same process* executing template Anc. A flow assertion of the form Prot(Source, Sink, Anc) specifies a *protected* flow: 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. Note that Prot assertions do not describe multiple-step, transitive flows over multiple processes, but rather describe direct, one-step flows between pairs of processes. However, a programmer can typically define policies using only a few Prot assertions by defining the Prot assertions over process templates that correspond to program points reached immediately before a process writes or reads any information.

Secrecy assertions specify what information flows from the original program must be prohibited in the instrumented program, while Prot assertions specify what flows from the original program must be allowed. A programmer can construct a policy that is inconsistent in that it specifies that some flows must be both prohibited and allowed in the original program. SWIM does *not* interpret such a policy by allowing some assertions to take precedence over others. Instead, SWIM identifies a minimal subset of the assertions that are inconsistent, and presents these to the programmer. This issue is discussed in further detail later in the section.

Although policies are defined over CSP programs, users may present policies in terms of the original C program. The user marks key program points with control-flow labels, and then constructs policies over the control-flow labels instead of CSP process templates. When SWIM translates a C program to its CSP representation, for each program point marked with a control-flow label, SWIM names the template corresponding to the program point with the text of the label. The policy originally given for the C program is then interpreted for the resulting CSP program.

EXA. 2. *The policy for the server in Fig. 1 can be expressed as the set of flow assertions:* $\mathsf{Secrecy}(W, W, \{P_1, P_3, P_5\}, A_1)$, $\mathsf{Prot}(W, P_3, A_1)$, $\mathsf{Prot}(P_5, R, \mathsf{init})$, *where* init *is a special process template that represents the root of the computation. The assertion* $\mathsf{Secrecy}(W, W, \{P_1, P_3, P_5\}, A_1)$ *specifies that a* Worker *process, which executes template* W*, may not send information to a different* Worker *process unless (i) the information flows through a process that executes a* Proxy *template* P_1*,* P_3*, or* P_5*, or (ii) the workers are created by the same* MPM *process that executed template* A_1*. The assertion* $\mathsf{Prot}(W, P_3, A_1)$ *specifies that a* Worker *process executing template* W *must be permitted 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*. Similarly, the assertion* $\mathsf{Prot}(P_5, R, \mathsf{init})$ *specifies that a* Proxy *process executing* P_5 *must be permitted to send data to the* Requester *executing* R*.*

In addition to flow assertions, SWIM takes a set of rules declaring which process templates denote processes that may be compromised. For the example in Fig. 1, SWIM takes a rule declaring that any Worker process may be compromised.[3]

[3]SWIM does not attempt to model all possible ways in which a process could be compromised. Instead, it generates "worst-case" constraints that assert that the process always tries to send information with its lowest possible label, and receive information with its highest possible label. Consequently, if a solution to the constraint system exists, it guarantees that a compromised process acting in such an "extremal" way will not be able to violate the desired information-flow policy.

From Programs and Policies to Instrumentation Constraints. Given a program and policy, SWIM generates a system of set constraints such that a solution to the constraints corresponds to an instrumentation of the program that satisfies the policy. The constraint system must assert that (i) the instrumented program uses the Flume API to manipulate labels in a manner allowed by Flume, and (ii) 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. SWIM 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_5 *to template* A_6*,* SWIM *generates the following four constraints:*

$$\mathsf{lab}_{A_6} \subseteq \mathsf{lab}_{A_5} \cup \mathsf{pos}_{A_6} \qquad \mathsf{pos}_{A_6} \subseteq \mathsf{pos}_{A_5} \cup \mathsf{creates}_{A_6}$$
$$\mathsf{lab}_{A_6} \supseteq \mathsf{lab}_{A_5} - \mathsf{neg}_{A_6} \qquad \mathsf{neg}_{A_6} \subseteq \mathsf{neg}_{A_5} \cup \mathsf{creates}_{A_6}$$

SWIM generates additional constraints to encode that the instrumented program does not allow flows that are specified by a Secrecy assertion, but allows all flows specified in a Prot assertion.

EXA. 4. *To enforce the flow assertion* $\mathsf{Secrecy}(W, W, \{P_1, P_3, P_5\}, A_1)$ *for the server in Fig. 1, the constraints will force the program to create a tag so that the undesired flows are prohibited. (How this occurs will become clearer in Exa. 5.) In §3.3.3 (Defn. 2), we define a set* Dist_{A_1} *of process templates such that if a distinct tag is created each time a process executes the template, then if two processes descend from distinct processes that executed* A_1 *and are marked with tags created at the template, then the two processes will have distinct tags.* SWIM *uses this information to encode the secrecy assertion as follows:*

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

SWIM *must guarantee that the flows described by the assertions* $\mathsf{Prot}(W, P_3, A_1)$ *and* $\mathsf{Prot}(P_5, R, \mathsf{init})$ *are permitted in the instrumented program. In §3.3.3 (Defn. 3), we define a set* Const_{A_1} *of process templates such that if a tag is created*

when a template Q in the set is executed, and two processes descend from the same process that executed A_1, and both processes carry a tag created at Q, then the processes carry the same tag. SWIM *uses this information to encode the two* Prot *assertions as follows:*

$$\begin{aligned} \mathsf{lab}_W &\subseteq \mathsf{lab}_{P_3} \cap \bigcup_{Q \in \mathsf{Const}_{A_1}} \mathsf{creates}_Q \\ \mathsf{lab}_{P_5} &\subseteq \mathsf{lab}_R \cap \bigcup_{Q \in \mathsf{Const}_{\mathsf{init}}} \mathsf{creates}_Q . \end{aligned}$$

From Instrumentation Constraints to DIFC Code. Solving constraint systems created by the method described above is an **NP**-complete problem. Intuitively, the complexity arises because such a constraint system may contain both positive and negative subset constraints and occurrences of set union. However, we have shown that if the constraint system has a solution, then it has a solution in which all variables have a value with no more tags than the number of Secrecy assertions in the policy [9, App. B]. Using this bound, SWIM 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, such as Yices [6], Z3 [16], or STP [8]. Such solvers implement decision procedures for decidable first-order theories, including the theory of bit vectors.

If the SMT solver determines that no solution exists for the bit-vector constraints, then it produces an *unsatisfiable core*, which is a minimal set of constraints that are unsatisfiable. In this case, the program and the policy assertions may be inconsistent. SWIM uses the unsatisfiable core identified by the solver to determine a minimal set of inconsistent flow assertions from the policy, and a minimal sub-program for which the assertions are inconsistent. SWIM 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 policy assertions that contributed to the unsatisfiable core.

On the other hand, if the SMT solver finds a solution to the bit-vector constraints, then SWIM translates this to a solution for the system of set constraints. Using the solution to the set-constraint system, SWIM then inserts DIFC code into the original program so that the label values of all processes over any execution of the program correspond to the values in the constraint solution. By construction, the resulting program satisfies the given information-flow policy.

EXA. 5. *The policy in Exa. 2 has one* Secrecy *flow assertion. Consequently, if the system of constraints generated for the program in Fig. 1 and the policy in Exa. 2 has a solution, then it has a solution over a set with one element.* SWIM *thus translates the system to an equisatisfiable bit-vector system over a set with a single element, and feeds the bit-vector system to an SMT solver. The solver determines that the bit-vector system has a solution, which is partially displayed below using set-values over the domain $\{\tau\}$:*

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	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

SWIM *uses the solution to generate the DIFC code highlighted in Fig. 1. In the solution,* $\mathsf{creates}_{A_5} = \{\tau\}$*, so* SWIM *inserts just before line A_5 (i.e., at A_4) a call to create a new tag t each time A_4 is executed. In the solution,* $\mathsf{lab}_{P_1} = \mathsf{pos}_{P_1} = \mathsf{neg}_{P_1} = \{\tau\}$*, so* SWIM *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* SWIM *inserts code just before P_5 (i.e., at P_4) to clear all members of the negative-capability set (i.e., t) from the label of the process. The final result is the full program given in Fig. 1.*

$\mathsf{Prog} := \mathsf{PROCVAR}_1 = \mathsf{Proc}_1 \ldots \mathsf{PROCVAR}_n = \mathsf{Proc}_n$
$\mathsf{Proc} := \mathsf{SKIP}$
$\quad |\mathsf{PROCVAR}$
$\quad |\ \mathsf{ChangeLabel}(LAB, LAB, LAB) \rightarrow \mathsf{PROCVAR}$
$\quad |\ \mathsf{CREATE}_\tau \rightarrow \mathsf{PROCVAR}$
$\quad |?\ \mathsf{PROC\_ID} \rightarrow \mathsf{PROCVAR}$
$\quad |!\ \mathsf{PROC\_ID} \rightarrow \mathsf{PROCVAR}$
$\quad |\mathsf{PROCVAR}_1 \ \square\ \mathsf{PROCVAR}_2$
$\quad |\mathsf{PROCVAR}_1 \ |||\ \mathsf{PROCVAR}_2$

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

3. DIFC INSTRUMENTATION

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

3.1 DIFC Programs

SWIM 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 described in [9, 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.

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 nondeterministically 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 decide 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 (STEP).
2. One process spawning another process (SPAWN).
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 $\mathsf{SPAWN}(\mathsf{p}, \mathsf{p}')$, 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 SWIM 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 nondeterministically 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 CREATE template can be bound in the namespaces of multiple processes. Furthermore, the set of tags created at such a template over an execution may be unbounded.
- $\mathsf{ChangeLabel}(L, M, N) \rightarrow \mathsf{PROCVAR}_1$: 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 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 [9, 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 an execution trace 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 in execution 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 transitive closure. Let $\mathsf{ShareAnc}(p, q, \mathsf{Anc},$

T) hold when processes p and q share an ancestor in Anc:

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

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

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

$\overrightarrow{P}'$ 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 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 of protection assertions $\mathcal{R}$, 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

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

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 SWIM 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

SWIM takes as input a program $\overrightarrow{P}$ and policy $\mathcal{F}$. To produce a program $\overrightarrow{P}'$ such that $\overrightarrow{P}' \models (\overrightarrow{P}, \mathcal{F})$, SWIM 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 SWIM were to use such a domain, then it could not produce a program that may create an unbounded set of tags over its execution. SWIM 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 $\overrightarrow{P}$, SWIM 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. 6. *The constraint variables used by* SWIM *are illustrated in Exa. 5. Consider the templates* A_5 *and* W*. The solution in Exa. 5 defines* $\mathsf{creates}_{A_5} = \{\tau\}$*. Thus* SWIM *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

SWIM 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, SWIM constrains 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, SWIM

generates the set of constraints SemCtrs defined as follows:

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

SemCtrs is defined by a function StepCtrs, which takes as input two template variables X and Y. 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 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 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:

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

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, Swim ensures 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:

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

Swim conjoins these constraints with the constraints generated from applying 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.

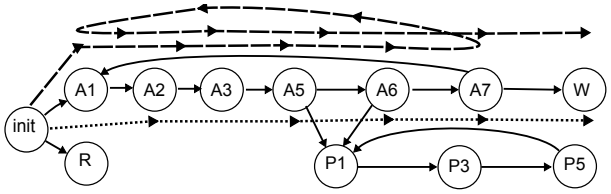

Figure 4: The spawn graph of the server from Fig. 1. The dotted and dashed paths indicate process executions that show that init cannot be used to create tags that isolate Workers.

3.3.3 Generating Policy Constraints

Swim must also generate constraints that ensure that the instrumented program satisfies the instrumentation conditions of §3.2.2. To do so, Swim generates constraints for each flow assertion in the policy.

First, suppose that Swim 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, Swim generates constraints that encode that, for processes $p \in$ Source and $q \in$ Sink that do not share an ancestor in Anc, information cannot flow from p to q solely through processes that are not in Declass.

To describe how Swim achieves this, we consider example executions that would violate the secrecy assertion. To describe these executions, we use the *spawn graph* of a program:

Def. 1. *For a* $\mathsf{CSP}_{\mathsf{DIFC}}$ *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. 7. *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$(W, W, \{P_1, P_3, P_5\}, A_1)$*. Suppose that* Swim *generated no constraints to ensure that the instrumented program followed this assertion.* Swim *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_2*,* A_3*,* 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_2*,* A_3*,* A_5*,* A_6*,* A_7*,* A_1*,* A_2*,* A_3*,* 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. 7 for an assertion Secrecy(Source, Sink, Declass, Anc), Swim could generate the constraint $\mathsf{lab}_{\mathsf{Source}} \not\subseteq \mathsf{lab}_{\mathsf{Sink}}$. However, this constraint may not allow Swim to find valid instrumentations of the program in important cases.

Exa. 8. *Suppose that for an assertion* Secrecy(Source, Sink, Declass, Anc)*,* Swim *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. 7, the resulting constraint, $\mathsf{lab}_W \not\subseteq \mathsf{lab}_W$*, is unsatisfiable, and* SWIM *would fail to instrument the server. However, if* SWIM *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 SWIM *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. 7 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*.*

SWIM captures the distinction between A_1 and init in Exa. 8 for a general secrecy assertion Secrecy(Source, Sink, Declass, Anc) by ensuring that there is either a tag identifier $\tau \in \mathsf{lab}_{\mathsf{Source}}$ such that $\tau \notin \mathsf{lab}_{\mathsf{Sink}}$ *or* that τ is 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, SWIM could thus weaken the constraint $\mathsf{lab}_{\mathsf{Source}} \not\subseteq \mathsf{lab}_{\mathsf{Sink}}$ 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 $p \in$ Source to a process $q \in$ 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, SWIM 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 [9, App. F] that this constraint is sufficient to ensure that the instrumented program satisfies the secrecy assertion.

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

EXA. 9. *Consider the server from Fig. 1 and the protection assertion* $\mathsf{Prot}(P_5, R, \mathsf{init})$ *that each* Proxy *executing* P_5 *must be able to send information to the* Requester *executing* R*. Suppose that* SWIM *generated no constraints to ensure that the program followed this assertion.* SWIM *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_2*,* A_3*,* 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 a communication from* p *to* q *would fail.*

Exa. 9 demonstrates that for assertion Prot(Source, Sink, Anc), if a tag in the label of $p \in$ Source is bound to an identifier τ, then for p to send information to $q \in$ 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 can communicate successfully, as demonstrated by the following example.

EXA. 10. *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* 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* Requester *process does not contain the label of all processes executing* P_3*. Thus a communication from a* Proxy *process to the* 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* Requester *executing* R*. Then the same tag would be in the labels of each* Proxy *process and the* Requester*. The key distinction between* A_1 *and* init *is that a* Proxy *and* Requester *may have distinct tags created at* A_1*, but cannot have distinct tags created at* init*.*

SWIM captures the distinction between init and A_1 in Exa. 10 for a general assertion Prot(Source, Sink, Anc) by strengthening the above constraint that $\mathsf{lab}_{\mathsf{Source}} \subseteq \mathsf{lab}_{\mathsf{Sink}}$. SWIM additionally ensures that if a tag in the label of $p \in$ 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 Prot(Source, Sink, Declass), the conditions on each tag identifier τ are expressed formally using $\mathsf{Const}_{\mathsf{Anc}}$ as:

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

Equivalently:

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

We prove in [9, 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 [9, 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 policy assertions. A solution to the conjunction of these constraints with the semantic constraints, $\varphi_{\mathsf{Tot}} \equiv \varphi_{\mathsf{Sem}} \wedge \varphi_{\mathsf{Pol}}$, 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, Swim 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 **NP**-complete in the number of terms in φ_{Tot}. We give a proof of hardness in [9, 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 [6] can typically solve large logical formulas very quickly. To apply an SMT solver, Swim 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}, Swim 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 [9, App. B].

Swim 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, Swim computes the subprogram and flow assertions that contributed constraints in the core, and presents these to the user as being in conflict. If the SMT solver determines that the constraint system is satisfiable, then it produces a solution to the system. Swim then 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 Swim 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 Swim rewrites P to $\mathsf{CREATE}_\tau \rightarrow P$. Now, let L, M, and N be the sets of tag identifiers in the constraint values for lab_X, pos_X, and neg_X. Then Swim rewrites P to $\mathsf{ChangeLabel}(L, M, N) \rightarrow P$. Swim can reduce the number of ChangeLabel templates generated by only generating such a template when a label or capability changes from that of a preceding P template in $G_{\overrightarrow{P}}$.

Swim 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 to reason about conditions in the original program code. *Swim* does not currently support such reasoning. However, our experiments, described in §4, indicate that in practice, Swim can successfully instrument real-world programs to handle real-world policies.

4. EXPERIMENTS

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

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

To examine these properties, we implemented Swim as an automatic tool[4] 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 [14].
3. The scanner module of ClamAV [4].
4. The OpenVPN client [19].

For each program module, we chose an information-flow policy from the literature [14, 23], 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 [18] program-analysis infrastructure for C, and the Yices SMT solver [6]. 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 discussed in §2. The original Apache code is represented by the non-underlined, unshaded code.

When we initially applied Swim, it determined that it could not instrument the MPM to enforce the supplied policy, and produced a minimal failing sub-program and sub-

[4]Available at `http://cs.wisc.edu/~wrharris/software/difc`

policy for diagnostic purposes. The fact that Swim could not instrument the MPM without manual changes should not be interpreted as a failure on the part of Swim. On the contrary, the fact that Swim could not instrument the program indicated that the program had to be restructured to allow for a correct instrumentation. Moreover, while the original MPM implementation is approximately 15,000 lines, Swim produced a minimal failing sub-program of only a few hundred lines, consisting mostly of statements that spawn processes and send or receive information between processes. The sub-policy contained only the Secrecy assertion that Worker processes must be isolated, and the Prot assertion that each Worker process must be able to send information to the Requester. By inspecting this sub-program and sub-policy, it was significantly easier to understand what manual edits we needed to perform to allow for a correct instrumentation. We manually added the underlined code in Fig. 1 to spawn proxy processes to mediate interprocess communication. We did not add any code that manipulated Flume labels explicitly. We then fed Swim the version of the MPM with proxies and the original policy, and Swim instrumented it correctly.

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 [14] is based on the software package MoinMoin Wiki [15], 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 Swim'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 [9, 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 Flume API calls that were similar to the original, manually inserted calls to the Flume API.

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.

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 1: Performance of Swim.

Inspired by [23], 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 [23]. 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 should 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 Swim 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, realistic policies. While not all programs could be instrumented to satisfy a desired policy without modification, when an instrumentation does exist, SWIM was able to find it each time.

For each application, we 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 SWIM to instrument the program. Col. "Num. Inst." gives the number of statements instrumented by SWIM. The results indicate that SWIM is a practical technique: SWIM is able to instrument large, real-world program modules in seconds. It is even fast enough 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 [21], HiStar [23], and Flume [14]. These systems all provide low-level mechanisms that allow an application programmer to implement an information-flow policy. SWIM complements these systems. By running program instrumented automatically by SWIM on on top of a DIFC operating system, a user obtains greater assurance of the end-to-end information-flow security of their application.

Our goals are shared by Efstathopoulos and Kohler [7], who have also explored the idea of describing a policy as declarations of allowed and prohibited information flows, for the Asbestos DIFC system. However, their work appears to have some significant limitations [7, Sec. 7] (emphasis ours):

> . . . developers are expected to produce sensible policy descriptions and our [instrumenter] is currently *unable to identify the configurations that are impossible to implement using IFC*. We would like to formalize the characteristics of policy descriptions that cannot be mapped to valid (and secure) label implementations so as to identify such cases and handle them accordingly (e.g. produce helpful, diagnostic error messages).

Our work goes beyond that of Efstathopoulos and Kohler by bringing more powerful formalisms to bear on the problem – in particular,

- the use of a constraint solver to determine what labels to use and where label-manipulation primitives should be placed,
- unsatisfiability as a formal criterion for when programs and policies are impossible to implement using only DIFC primitives.

We apply these formalisms to obtain multiple benefits. First, from an unsatisfiable core of constraints, we can provide help in diagnosing failures by exhibiting the subprogram of the original program and the subset of the policy declarations that contributed constraints to the unsatisfiable core. Second, the technique of [7] relies on the "floating-label" semantics of Asbestos, in which communications can implicitly change the labels of processes. The work presented in [13] shows that a system with such a semantics enables high-throughput leaks, while a system such as Flume, in which labels are explicitly manipulated by each process, is provably free of such leaks.

Harris et al. [10] 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 [13] use CSP to reason about the Flume OS, not applications running atop Flume.

Resin [22] 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 [22].

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 [17, 20]. Jif has been used to implement several real-world applications with strong security guarantees (e.g. [3, 5, 11]), but these programs are written from scratch in Jif. Automatic techniques can partition a Jif web application between its client and server [3]. Jif requires the programmer to define a bounded set of principals, and provide annotations that specify which code objects should be able to read and write data in terms of these principals. In contrast, SWIM can reason about policies that require an unbounded set of principals to have different access capabilities. Furthermore, SWIM automatically infers the set of tags that represent the information-flow capabilities of principals, along with inferring the code that manipulates these tags.

Aspect-oriented programming (AOP) breaks program logic down into distinct parts (called concerns) [12]. AOP deals with concerns (called crosscutting concerns) that span multiple abstractions in a program. Logging exemplifies a crosscutting concern because a logging strategy necessarily affects every single logged part of the system. An aspect can alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches). The process of adding additional behavior to the base code is called aspect weaving.

At an abstract level, policy weaving is a special case of aspect weaving for security. In this work, we describe techniques that automate the entire process of aspect weaving for security. Moreover, whereas conventional AOP systems use syntactically-oriented mechanisms and allow aspect code to be placed in a rather limited set of locations, our approach is based on programming-language and policy semantics, and allows policy-enforcement primitives to be placed in essentially arbitrary locations. The policy-weaving techniques described in the proposal can potentially be applied to other crosscutting concerns, such as logging and failure handling, to improve standard AOP.

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 sat-

isfy 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.

To date, we have worked with a very simple policy language in which policies consist of Secrecy and Prot assertions. Future work will explore more sophisticated policy languages for a broader set of policy issues. We will also investigate other approaches to handling inconsistent policies.

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

[7] P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. *SIGOPS Oper. Syst. Rev.*, 42(4):301–313, 2008.

[8] V. Ganesh and D. Dill. A decision procesure for bit-vectors and arrays. In *CAV*, 2007.

[9] W. R. Harris, S. Jha, and T. Reps. DIFC programs by automatic instrumentation. `http://cs.wisc.edu/~wrharris/publications/tr-1673.pdf`, 2010.

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

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

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

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

[14] 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.

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

[16] L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

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

[18] 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.

[19] OpenVPN. `http://www.openvpn.net`.

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

[21] 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.

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

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