Verifying Information Flow Control Over Unbounded Processes

William R. Harris[1], Nicholas A. Kidd[1], Sagar Chaki[2], Somesh Jha[1], and Thomas Reps[1,3]

[1] University of Wisconsin; {wrharris, kidd, jha, reps}@cs.wisc.edu
[2] Soft. Eng. Inst.; Carnegie Mellon University; chaki@sei.cmu.edu
[3] GrammaTech Inc.

Abstract. *Decentralized Information Flow Control* (DIFC) systems enable programmers to express a desired DIFC policy, and to have the policy enforced via a reference monitor that restricts interactions between system objects, such as processes and files. Past research on DIFC systems focused on the reference-monitor implementation, and assumed that the desired DIFC policy is correctly specified. The focus of this paper is an automatic technique to verify that an application, plus its calls to DIFC primitives, does indeed correctly implement a desired policy. We present an abstraction that allows a model checker to reason soundly about DIFC programs that manipulate potentially unbounded sets of processes, principals, and communication channels. We implemented our approach and evaluated it on a set of real-world programs.

1 Introduction

Decentralized Information Flow Control (DIFC) systems [1–4] allow application programmers to define their own DIFC policies, and then to have the policy enforced in the context of the entire operating system. To achieve this goal, DIFC systems maintain a mapping from OS objects (processes, files, etc.) to *labels*—sets of atomic elements called *tags*. Each process in the program creates tags, and gives other processes the ability to control the distribution of the process's data by collecting and discarding tags. The DIFC runtime system monitors all inter-process communication, deciding whether or not a requested data transfer is allowed based on the labels of system objects.

Example 1. Consider the diagram in Fig. 1 of a web server that handles sensitive information. A `Handler` process receives incoming HTTP requests, and spawns a new `Worker` process to service each request. The `Worker` code that services the request may not be available for static analysis, or may be untrusted. The programmer may wish to enforce a *non-interference policy* requiring that information pertaining to one request — and thus localized to one `Worker` process — should never flow to a different `Worker` process.

DIFC mechanisms are able to block any communication between OS objects. Thus, in addition to ensuring that the use of DIFC mechanisms implements a

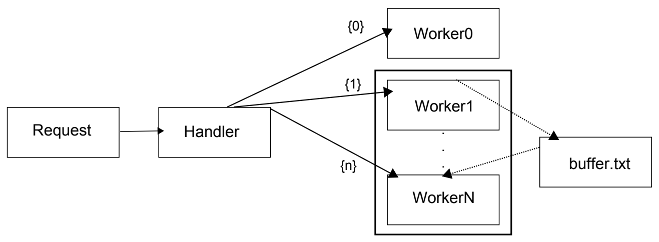


Fig. 1. An inter-process diagram of a typical web server.

desired security policy, the programmer must also ensure that retrofitting an existing system with DIFC primitives does not negatively impact the system's functionality. In Ex. 1, the desired functionality is that the `Handler` must be able to communicate with each `Worker` at all times. An overly restrictive implementation could disallow such behaviors. Ex. 1 illustrates a tension between security and functionality: a naïve system that focuses solely on functionality allows information to flow between all entities; conversely, the leakage of information can be prevented in a way that cripples functionality.

Our goal is to develop an automatic method to ensure that security policies and application functionality are simultaneously satisfied. Our approach to this problem is to leverage progress in model checkers [5, 6] that check concurrent programs against temporal logic properties (e.g., linear temporal logic). However, the translation from arbitrary, multiprocess systems to systems that can be reasoned about by model checkers poses key challenges due to potential unboundedness along multiple dimensions. In particular, the number of processes spawned, communication channels created, and label values used by the reference monitor are unbounded. However, current model checkers verify properties of models that use bounded sets of these entities. To resolve this issue, we propose a method of abstraction that generates a model that is a sound, and in practice precise, approximation of the original system in the sense that if a security or functionality property holds for the model, then the property holds for the original program. Our abstraction applies the technique of *random isolation* [6] to reason precisely about unbounded sets of similar program objects.

The contributions of this work are as follows:

1. We present a formulation of DIFC program execution in terms of transformations of logical structures. This formulation allows a natural method for abstracting DIFC programs to a bounded set of structures. It also permits DIFC properties of programs to be specified as formulas in first-order logic. To our knowledge, this is the first work on specifying a formal language for such policies.
2. We present a formulation of the principle of random isolation [6] in terms of logical structures. We then demonstrate that random isolation can be applied to allow DIFC properties to be checked more precisely.

3. We implemented a tool that simulates the abstraction of logical structures in C source code and then checks the abstraction using a predicate-abstraction-based model checker.
4. We applied the tool to check properties of several real-world programs. We automatically extracted models of modules of Apache [7], FlumeWiki [3], ClamAV [8], and OpenVPN [9] instrumented with our own label-manipulation code. Verification took a few minutes to less than 1.25 hours.

While there has been prior work [10, 11] on the application of formal methods for checking properties of actual DIFC systems, our work is unique in providing a method for checking that an *application* satisfies a DIFC correctness property under the rules of a given DIFC system. Our techniques, together with the recent verification of the Flume reference-monitor implementation [11], provides the first system able to (i) verify that the program adheres to a specified DIFC policy, and (ii) verify that the policy is enforced by the DIFC implementation.

The rest of the paper is organized as follows: §2 describes Flume, an example DIFC system for which we check applications. §3 gives an informal overview of our techniques. §4 gives the technical description. §5 describes our experimental evaluation. §6 discusses related work.

2 A Flume Primer

Our formulation is based most closely on the Flume [3] DIFC system; however, our abstraction techniques should work with little modification for most DIFC systems. We briefly discuss the Flume datatypes and API functions provided by Flume, and direct the reader to [3] for a complete description.

- **Tags & 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**. A *positive capability* t^+ allows a process to add tag t to the label of an OS object. A *negative capability* t^- allows a process to remove 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 may be claimed by at most *one* process, after which they are used like ordinary file descriptors.

For each process, Flume maintains a secrecy label, an integrity label, and a capability set. In this work, we only consider secrecy labels, and leave the modeling of integrity labels as a direction for future work. The monitor forbids a process p with label l_p to send data over endpoint e with label l_e unless $l_p \subseteq l_e$. Likewise, the monitor forbids a process p' to receive data from endpoint e' 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 a binary to execute, (ii) a set of endpoints that the new process may access from the beginning of execution, (iii) an initial label, (iv) and an initial capability set, which must be a subset of that of the spawning process. An example usage of `spawn` is given in Fig. 2.

```
  void Handler() {
1.    Label lab;
2.    int data;
3.    while (*) {
4.        Request r = get_next_http_request();
5.        data = get_data(r);
7.        lab = create_tag();
8.        Endpoint e0, e1;
9.        create_channel(&e0, &e1);
10.       spawn("/usr/local/bin/Worker", {e1}, lab, {}, r);
11.       data = recv(claim_endpoint(e0)); }}
```

Fig. 2. Flume pseudocode for a server that enforces the same-origin policy.

Example 2. The pseudocode in Fig. 2 enforces non-interference between the `Worker` processes from Fig. 1. The `Handler` perpetually polls for a new HTTP request, and upon receiving one, it spawns a new `Worker` process. To do so, it (i) has Flume create a new tag, which it stores as a singleton label value in label-variable `lab` (line 7), (ii) has Flume create a new channel (line 9), and (iii) then launches the `Worker` process (line 10), setting its initial secrecy label to `lab`—not giving it the capability to add or remove the tag in `lab` (indicated by the {} argument)—and passing it one end of the channel to communicate with the `Handler`. Because the `Handler` does not give permission for other processes to add the tag in `lab`, no process other than the `Handler` or the new `Worker` can read information that flows from the new `Worker`.

3 Overview

The architecture of our system is depicted in Fig. 3. The analyzer takes as input a DIFC program and a DIFC policy. First, the program is (automatically) extended with instrumentation code that implements *random-isolation* semantics (§3.3). Next, *canonical abstraction* (§3.2) is performed on the rewritten program to generate a finite-data model. Finally, the model and the DIFC policy are given as input to the concurrent-software model checker Copper, which either verifies that the program adheres to the DIFC policy or produces a (potentially spurious) execution trace as a counterexample. We now illustrate each of these steps by means of examples.

3.1 Concrete Semantics

Program states are represented using *first-order logical structures*, which consist of a collection of individuals, together with an interpretation for a finite

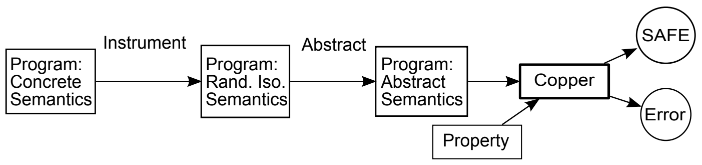

Instrument
Abstract
Program: Concrete Semantics
Program: Rand. Iso. Semantics
Program: Abstract Semantics
Copper
Property
SAFE
Error

Fig. 3. System architecture.

vocabulary of finite-arity relation symbols. An *interpretation* is a truth-value assignment for each relation symbol and appropriate-arity tuple of individuals. For DIFC systems, these relations encode information such as:

1. "Label variable x does (or does not) contain tag t in its label."
2. "Process p has (or has not) sent information to process q."

Tab. 1 depicts structures as they are transformed by program statements. The convention used is that every boxed node represents a tag individual, and every circled node represents a label individual. The name of an individual may appear outside of the circle. An arrow from an identifier to a circle individual denotes that a particular unary relation holds for the individual, and an arrow between nodes denotes that a binary relation holds for the individuals. A dotted arrow indicates that it is unknown whether a given tuple of a relation does or does not hold, and the relationship is said to be *indefinite*. Otherwise, the relationship is said to be *definite* (i.e., definitely holds or definitely does not hold). A doubled box indicates a *summary* individual, which represents one or more concrete individuals in abstracted structures. The value of the unary relation iso is written inside the individuals in the diagrams for random-isolation semantics. A program statement transforms one structure to another, possibly by adding individuals to the structure or altering the values of relations. State properties are specified as logical formulas.

Example 3. The top row, left column of Tab. 1 gives an example of how one concrete state is transformed into the next state by execution of the statement `lab = create_tag()`. Suppose that the statement has been executed twice previously, introducing tag individuals t and u, where u is a member of the label m. This containment is encoded by a relation R_{Tag}, denoted in the figure by an arrow from m to t. The next execution of the statement creates a new tag v and relates the label m to v while ending the relationship between m and u.

3.2 Canonical Abstraction

Unbounded sets of concrete structures can be abstracted into bounded sets of abstract structures using canonical abstraction with respect to some set of unary relations $\mathcal{A}$ [12]: each concrete structure S is mapped to an abstract structure $S^{\#}$ that has exactly one individual for each combination of unary relations in $\mathcal{A}$. If multiple concrete individuals that map to the same canonical individual yield

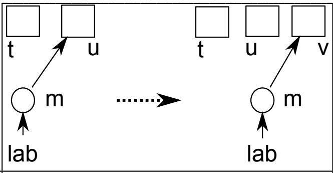
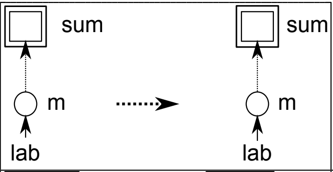
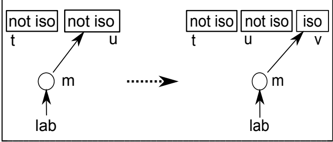
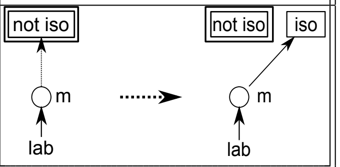
	Concrete Semantics	Abstract Semantics
Standard		
Random Isolation		

Table 1. Illustration of the standard and random-isolation semantics of the statement `lab := create_tag()` and their corresponding abstract semantics.

different values in some relation $R \notin \mathcal{A}$, then the abstract structure records that the abstract individual may or may not be in the relation R.

Example 4. Consider the top row of Tab. 1. Assume that every tag created belongs to the same relations in $\mathcal{A}$. Thus all tag individuals will be mapped under canonical abstraction into a single abstract summary tag individual, sum. This can introduce imprecision. Suppose that in a concrete structure, $R_{Tag}(m, u)$ holds before the latest execution of `create_tag`, at which point it no longer holds and $R_{Tag}(m, v)$ holds. In the abstraction of this structure, tags t, u, and v are all represented by *sum*. Thus, after the second execution of the statement `create_tag`, sum represents all of label m's tags, but also represents tags that are not elements of m. This is reflected in Tab. 1 by a dotted arrow from the label m to the summary tag individual.

3.3 Random Isolation

Ex. 4 demonstrates that abstraction over the unary relations of a program state is insufficiently precise to prove interesting DIFC properties. We thus use random isolation [6] to reason about an individual from a set, and to generalize the properties proved about the individual object to a proof about all individuals in the set. Random isolation can be formulated as introducing a special unary relation iso in structures. When freshly allocated individuals are created and relations over individuals are updated, the invariant is maintained that iso holds for at most one individual. Furthermore, if iso ever holds for an individual u, then it must continue to hold for u for the remainder of the program's execution. The relation iso can be used to increase the precision of reasoning about an individual in an abstract structure.

Example 5. Consider the state transformation given in the bottom row, left column of Tab. 1. When the program is executed under random-isolation semantics, the call to `create_tag` non-deterministically chooses whether iso holds for the newly created tag. Tab. 1 illustrates the case in which the present call returns a tag for which iso does hold. Now consider the bottom row, right column of Tab. 1, in which the relations used to abstract program states include iso. There is now a definite relationship between the label and the most recently allocated tag. This definite relationship may allow for stronger claims to be made about a structure when properties of the structure are checked.

4 Formal Definition of Abstraction

4.1 Inputs

Subject programs. To simplify the discussion, we assume a simple imperative language L_{Lab} in which programs are only able to manipulate DIFC labels, send and receive data over channels, and spawn new processes. The full grammar for such a language is given in [13]. The semantics of a program in L_{Lab} can be expressed by encoding program states as logical structures with interpretations in 2-valued logic. A 2-valued logical structure S is a pair $\langle U^S, \iota \rangle$, accompanied by a vocabulary of relations $\mathcal{P}$ and constants $\mathcal{C}$. The set U^S is the universe of individuals of S, and for $p \in \mathcal{P}$ of arity k and every tuple $(u_1, u_2, \ldots, u_k) \in (U^S)^k$, the interpretation ι maps $p(u_1, u_2, \ldots, u_k)$ to a truth value: 0 or 1. The interpretation ι also maps each constant in $\mathcal{C}$ to an individual.

Let $\mathcal{A} \subseteq \mathcal{P}$ be the set of unary *abstraction relations*. For a 2-valued structure S, S' is the *canonical abstraction* of S with respect to $\mathcal{A}$ if S' is a 3-valued structure in which each individual in the universe of S' corresponds to a valuation of the relations in $\mathcal{A}$. Each element in S then maps under an *embedding function* α to the element that represents its evaluation under the relations in $\mathcal{A}$. By construction, for $\alpha(u) \in U^{S'}$ and $R \in \mathcal{A}$, it is the case that $R(\alpha(u)) \in \{0, 1\}$. However, it may be the case that for some individual $u \in U^{S'}$, there exist $u_1, u_2 \in \alpha^{-1}(u)$ and a relation $p \in \mathcal{P}$ such that $p(\ldots, u_1, \ldots) = 0$ and $p(\ldots, u_2, \ldots) = 1$. It is then the case that $p(\ldots, u, \ldots) = 1/2$ where $1/2$ is a third truth value that indicates the absence of information, or uncertainty about the truth of a formula. The truth values are partially ordered by the precision ordering $\sqsubseteq$ defined as $0 \sqsubset 1/2$ and $1 \sqsubset 1/2$. Values 0 and 1 are called *definite* values; $1/2$ is called an *indefinite* value. If φ is a closed first-order logical formula, then let $[\![\varphi]\!]_2^S$ denote the 2-valued truth value of φ for a 2-valued structure S, and let $[\![\varphi]\!]_3^{S'}$ denote its 3-valued truth value for a 3-valued structure S'. For a more complete discussion of the semantics of 3-valued logic, see [12].

By the Embedding Theorem of Sagiv et al. [12], if S is a 2-valued structure, S' is the canonical abstraction of S with embedding function α, Z is an assignment that has a binding for every free variable in φ, and $[\![\varphi]\!]_3^{S'}(Z) \neq 1/2$, then it must be the case that $[\![\varphi]\!]_2^S(Z) = [\![\varphi]\!]_3^{S'}(\alpha \circ Z)$. In other words, any property that has a definite value in S' must have the same definite value in all S that abstract to S'.

In the context of label programs, individuals correspond to process identifiers, per-process variables, tags, channels, and endpoints. Relations encode the state of the program at a particular program point. The constants represent information about the currently executing program statement. Let $U^S = P \cup L \cup T \cup C \cup E$, where P is the set of process identifiers, M is the set of labels, T is the set of tags created during execution, C is the set of channels created during execution, and E is the set of endpoints created during execution.

We now consider a fragment of the relations and constants used in modeling. The complete sets of both the relations and constants are given in [13]:

- $\{\mathsf{isproc}(u), \mathsf{islabel}(u), \mathsf{istag}(u), \mathsf{ischannel}(u), \mathsf{isendp}(u)\}$ denote membership in each of the respective sets. These are the "sort" relations, denoted by Sorts. Each individual has exactly one sort.
- $R_x(u)$ is a unary relation that is true iff the label or endpoint u corresponds to program variable x.
- $R_P(u)$ is a unary relation that is true iff the process identified by u began execution at program point P.
- $R_{Tag}(u, t)$ is a binary relation that is true iff u is a label and t is a tag in the label of u.
- $R_{Chan}(e, c)$ is a binary relation that is true iff e is an endpoint of channel c.
- $R_{Owns}(p, u)$ is a binary relation that is true iff p is a process id and u is a label or endpoint that belongs to a variable local to p.
- $R_{Label}(u_1, u_2)$ is a binary relation that is true iff u_1 is a process identifier or an endpoint and u_2 is the label of u_1.
- For every set of entry points $\mathcal{G}$, there is a binary relation $R_{Flow:\mathcal{G}}(u_1, u_2)$ that is true iff u_1 and u_2 are process ids and there has been a flow of information from u_1 to u_2 only through processes whose entry points are in $\mathcal{G}$.
- $R_{Blocked}(p_1, p_2)$, a binary relation that is true iff p_1 and p_2 are process ids and there has been a flow of information from p_1 to p_2 that was blocked.

We consider the fragment of the vocabulary of constants $\mathcal{C} = \{\mathsf{cur}_p, \mathsf{cur}_{lab}, \mathsf{cur}_+, \mathsf{cur}_-, \mathsf{new}_t\}$. These denote the id, label, positive-capability, and negative-capability of the process that is to execute the next statement, along with the newest tag allocated. For a program with a process p designated as the first process to execute starting at program point P, the initial state of the program is the logical structure: $\langle \{p_{id}, p_{lab}, p_+, p_-\}, \iota \rangle$ where ι is defined such that each individual is in its sort relation, p_{id} is related to its entry point P, p_{id} is related to its label and capabilities, and the constants that denote the current process, its label, and its capabilities are mapped to p_{id}, p_{lab}, and p_+, p_- respectively.

To execute, the program non-deterministically picks a process, say q_{id}, and updates ι to map $cur_{id}, cur_{lab}, cur_+$, and cur_- to the id, label, and capabilities of q_{id}. The program then executes the next statement of process q_{id}. The statement transforms the relations over program state as described by the action schemas in [13]. We provide a few of the more interesting schemas in Fig. 4 as examples. For clarity in presenting the action schemas, we use the meta-syntax $\mathsf{if}\ \varphi_0\ \mathsf{then}\ \varphi_1\ \mathsf{else}\ \varphi_2$ to represent the formula $(\varphi_0 \rightarrow \varphi_1) \land (\neg\varphi_0 \rightarrow \varphi_2)$. Additionally, we define $\mathsf{subset}(x, y)$ to be true iff the label x is a subset of the label y:

$$\mathsf{subset}(x, y) \equiv \forall t : R_{Tag}(x, t) \rightarrow R_{Tag}(y, t) \qquad (1)$$

Along with transforming relations, some program statements have the additional effect of expanding the universe of individuals of the structure. In particular:

– `create_channel` adds new endpoint individuals u_{e0}, u_{e1} and a new channel individual u_c to the universe. It redefines the interpretation to add each of these individuals to the appropriate sort relations.
– `create_tag` adds a new tag individual u_t and similarly defines the interpretation to add this tag to its sort relation.
– `spawn` adds a new process id p_{id} and new labels p_{lab}, p_+, p_- that represent the label, positive capability, and negative capability of the process, respectively. It redefines the interpretation to add each of these individuals to their respective sorts.

Fig. 4 defines formally the action schema for the following two actions:

– `send(e);` attempts to send data from the current process to the channel that has `e` as an endpoint. The action potentially updates both the set of all flow-history relations and the blocked-flow relation. To update a flow-history relation $R_{Flow:\mathcal{G}}(u_1, u_2)$, the action checks if u_1 represents the id of the current process. If so, it takes f, the endpoint in u_1 to which the variable e maps, and checks if f is an endpoint of the channel u_2. If so, it checks if the label of u_1 is a subset of that of the endpoint of f and if so, adds (u_1, u_2) to the flow-history relation. Otherwise, the relation is unchanged.
  To update relation $R_{Blocked}(p_1, p_2)$, let f be the endpoint that belongs to u_1 and mapped by variable `e`. If f is an endpoint of a channel for which the other endpoint is owned by p_2, and the label of p_1 is not a subset of the label of f, then $R_{Blocked}(p_1, p_2)$ is updated. Otherwise, the relation is unchanged.
– `l := create_tag();` creates a new tag and stores it in the variable l. This updates the relation R_{Tag}. To update the value of the entry $R_{Tag}(u, t)$, the action checks if u represents a label belonging to the current process and if the variable l maps to u. If so, then $R_{Tag}(u, t)$ holds in the post-state if and only if t is the new tag. Otherwise, the relation R_{Tag} is unchanged.

Specifications. DIFC specifications can be stated as formulas in first-order logic. The following specifications are suitable for describing desired DIFC properties for programs written for DIFC systems.

– $\mathsf{NoFlowHistory}(P, Q, D) = \forall p, q : (R_P(p) \wedge R_Q(q) \wedge p \neq q) \rightarrow \neg R_{Flow:(\mathcal{G}-\{D\})}(p, q)$. For program points P, Q, D, this formula states that no process that begins execution at P should ever leak information to a different process that begins execution at Q unless it goes through a process in D. Intuitively, this can be viewed as a *security property.*
– $\mathsf{DefiniteSingleStepFlow}(P, Q) = \forall p, q : (R_P(p) \wedge R_Q(q)) \rightarrow \neg R_{Blocked}(p, q)$. For program points P and Q, this formula states that whenever a process that begins execution in P sends data to a process that begins execution in Q, then the information should not be blocked. Intuitively, this can be viewed as a *functionality property.*

Statement	Update Formula
`send(e);`	$R'_{Flow:\mathcal{G}}(u_1, u_2) = \mathsf{isproc}(u_1) \land \mathsf{ischan}(u_2) \land (u_1 = cur_{id}$ $\land(\exists f, m : R_e(f) \land R_{Owns}(u_1, f)$ $\land R_{Chan}(f, u_2) \land R_{Label}(f, m)$ $\land\ (\exists s : R_{Flow:\mathcal{G}}(s, u_1))$ $\land\ \mathsf{subset}(cur_{lab}, m))) \lor R_{Flow:\mathcal{G}}(u_1, u_2)$ $R'_{Blocked}(p_1, p_2) = \mathsf{isproc}(p_1) \land \mathsf{isproc}(p_2) \land (u_1 = cur_{id}$ $\land(\exists c, f, g, m : R_{Owns}(f) \land R_e(f)$ $\land R_{Chan}(f, c) \land R_{Chan}(g, c)$ $\land R_{Owns}(p_2, g) \land R_{Label}(f, m)$ $\land\neg \mathsf{subset}(cur_{lab}, m)))$ $\lor R_{Blocked}(p_1, p_2)$
`l := create_tag();`	$R'_{Tag}(u, t) = \mathsf{islabel}(u) \land \mathsf{istag}(t)$ $\land$ if $R_{Owns}(\mathsf{cur}_{id}, u) \land R_l(u)$ then $t = \mathsf{new}_t$ else $R_{Tag}(u, t)$

Fig. 4. Example action schemas for statements in L_{Lab}. Post-state relations are denoted with primed variables. Each post-state tuple of a relation R is expressed in terms of values of pre-state tuples. Post-state relations that are the same as their pre-state counterparts are not shown above.

Abstraction. The abstract semantics of a program in L_{Lab} can now be defined using 3-valued structures. Let P be a program in L_{Lab}, with a set of program variables $\mathcal{V}$ and a set of program points $\mathcal{L}$. To abstract the set of all concrete states of P, we let the set of abstraction relations $\mathcal{A}$ be $\mathcal{A} = \mathsf{Sorts} \cup \{R_x | x \in \mathcal{V}\} \cup \{R_P | P \in \mathcal{L}\}$.

By the Embedding Theorem [12], a sound abstract semantics is obtained by using exactly the same action schemas that define the concrete semantics, but interpreting them in 3-valued logic to obtain transformers of 3-valued structures.

4.2 Checking Properties Using Random Isolation

A simple example suffices to show that the canonical abstraction of a structure S based on the set of relations $\mathcal{A}$ is insufficiently precise to establish interesting DIFC properties of programs.

Example 6. Consider again the server illustrated in Fig. 1. In particular, consider a *concrete* state of the program with structure S in which n Worker processes have been spawned, each with a unique tag t_k for process k. In this setting, when a Worker with label u_i attempts to send data to a different Worker that can read data over a channel with an endpoint u_j where $i \neq j$, then $\mathsf{subset}(u_i, u_j) = 0$. Thus, no information can leak from one Worker to another. However, an analysis of the *abstract* states determines soundly, but imprecisely, that such a program might not uphold the specification $\mathsf{NoFlowHistory}(\mathsf{Worker}, \mathsf{Worker}, \emptyset)$. Let S' be the canonical abstraction of S based on $\mathcal{A}$. Under this abstraction, all tags in S are merged into a single abstract tag individual t in S'. Thus, for any process p,

$R_{Tag}(p,t) = 1/2$, and subset yields $1/2$ when comparing the labels of any process and any endpoint. Thus, if a Worker attempts to send data over a channel used by another Worker, the analysis determines that the send *might* be successful and thus that data may be leaked between separate Worker processes.

Intuitively, the shortcoming in Ex. 6 arises because the abstraction collapses information about the tags of all processes into a single abstract individual. However, random isolation can prove a property for one tag individual t non-deterministically distinguished from the other tags, and then soundly infer that the property is true of all tags individuals. We first formalize this notion by stating and proving the principle of random isolation in terms of 3-valued logic. We then examine how the principle can be applied to DIFC program properties. The proof requires the following lemma:

Lemma 1. *Let $\varphi[x]$ be a formula that does not contain the relation* iso*, and in which x occurs free. Let S be a 2-valued structure. For $u \in U^S$, let ρ_u map S to a structure that is identical to S except that it contains a unary relation* iso *that holds only for element u. Let α perform canonical abstraction over the set of unary relations $\mathcal{A} \cup \{\mathsf{iso}\}$. Then*

$$[\![\forall x : \varphi[x]]\!]_2^S \sqsubseteq \bigsqcup_{u \in U^S} [\![\forall x : iso(x) \rightarrow \varphi[x]]\!]_3^{\alpha(\rho_u(S))}$$

Proof. See [13]. □

The benefits of random isolation stem from the following theorem, which shows that when checking a universally quantified formula $\forall x : \varphi[x]$, one needs to check whether the weaker formula $\forall x : iso(x) \rightarrow \varphi[x]$ holds.

Theorem 1. *For a program P, let $\mathcal{T}$ be the set of all logical structures that are reachable under the standard, concrete semantics of P, and let $\mathcal{U}$ be the set of all abstract 3-valued structures that are reachable under the 3-valued interpretation of the concrete semantics after the random-isolation transformation has been applied to P. Let $\varphi[x]$ be a formula that does not contain the relation iso. Then*

$$\bigsqcup_{S \in \mathcal{T}} [\![\forall x : \varphi[x]]\!]_2^S \sqsubseteq \bigsqcup_{S^\# \in \mathcal{U}} [\![\forall x : iso(x) \rightarrow \varphi[x]]\!]_3^{S^\#} \tag{2}$$

Proof. Let $S \in \mathcal{T}$ be a state reachable in the execution of P. Let A be defined on a two-valued structure S as $A(S) = \bigcup_{u \in U^S} \{\alpha(\rho_u(S))\}$. By the soundness of the abstract semantics, it must be the case that for $S' \in A(S)$, there exists some $S^\# \in \mathcal{U}$ such that S' embeds into $S^\#$. Thus, by the Embedding Theorem [12],

$$\bigsqcup_{u \in U^S} [\![\forall x : iso(x) \Rightarrow \varphi(x)]\!]_3^{\alpha(\rho_u(S))} \sqsubseteq \bigsqcup_{S^\# \in \mathcal{U}} [\![\forall x : iso(x) \Rightarrow \varphi(x)]\!]_3^{S^\#}$$

and thus by Lem. 1, we have

$$[\![\forall x : \varphi(x)]\!]_2^S \sqsubseteq \bigsqcup_{S^\# \in \mathcal{U}} [\![\forall x : iso(x) \Rightarrow \varphi(x)]\!]_3^{S^\#}$$

Eqn. (2) follows from properties of $\sqcup$ and the soundness of the abstract semantics [12]. $\square$

Example 7. Consider again the example of the server with code given in Fig. 2 checked against the specification $\mathsf{NoFlowHistory}(\mathsf{Worker}, \mathsf{Worker}, \emptyset)$. Let the server code execute under random-isolation semantics with isolation relations $\mathsf{isoproc}$ and isotag. We want to verify that every state reachable by the program satisfies the formula $\mathsf{NoFlowHistory}(P, Q, D)$. Thm. 1 can be applied here in two ways:

1. One can introduce a unary relation $\mathsf{isoproc}$ that holds true for exactly one process id and then check

$$\forall p : isoproc(p) \rightarrow \forall q : ((R_P(p) \wedge R_Q(q)) \rightarrow \neg R_{Flow:\mathcal{G}-\{\mathcal{D}\}}(p, q))$$

   Intuitively, this has the effect of checking only the isolated process to see if it can leak information.
2. Consider instances where a flow relation $R_{Flow:\mathcal{G}}$ is updated on a `send` from the isolated process. Information will only be allowed to flow from the sender if the label of the sender is a subset of the label of the endpoint. The code in Fig. 2 does not allow this to happen, but the abstraction of the (ordinary) concrete semantics fails to establish that the flow is definitely blocked (illustrated in Ex. 6).
   By Thm. 1, one can now introduce a unary relation isotag that holds for at most one tag and instead of checking subset as defined in Eqn. (1), check the formula: $\forall t : \mathsf{isotag}(t) \rightarrow (R_{Tag}(x, t) \Rightarrow R_{Tag}(y, t))$. When the tag is in the sender's label, the abstract structure encodes the fact that the tag is held by exactly one process: the isolated sender. Thus the abstraction of the random-isolation semantics is able to establish that the flow is definitely blocked.

5 Experiments

There is an immediate correspondence between the operations defined in the grammar of L_{Lab} and the API of the DIFC system Flume. We took advantage of a close correspondence between abstraction via 3-valued logic and predicate abstraction (details can be found in [13]) and modeled the abstraction of the random-isolation semantics in C code via a source-to-source translation tool implemented with CIL [14], a front-end and analysis framework for C (see [13]). The tool takes as input a program written against the Flume API that may execute using bounded or unbounded sets of processes, tags, and endpoints. Our experiments demonstrate that information-flow policies for real-world programs used in related work [1–4] can often be expressed as logical formulas over structures that record DIFC state; that these policies can be checked quickly, and proofs or violations can be found for systems that execute using bounded sets of processes and tags; and that these policies can be checked precisely, albeit in significantly more time, using random isolation to find proofs or violations for programs that execute using unbounded processes and tags.

Program	Size (LOC)	# Procs. (runtime)	Property	Result	Time
FlumeWiki	110	unbounded	Correct	safe	1h 9m 16s
			Interference	possible bug	37m 53s
Apache	596	unbounded	Correct	safe	1h 13m 27s
			Interference	possible bug	18m 30s
ClamAV	3427	2	Correct	safe	7m 55s
			NoRead	possible bug	3m 25s
			Export	possible bug	3m 25s
OpenVPN	29494	3	Correct	safe	2m 17s
			NoRead	possible bug	2m 52s
			Leak	possible bug	2m 53s

Table 2. Results of model checking.

We applied the tool to three application modules—the request handler for FlumeWiki, the Apache multi-process module, and the scanner module of the ClamAV virus scanner—as well as the entire VPN client, OpenVPN. For each program, we first used the tool to verify that a correct implementation satisfied a given DIFC property. We then injected faults into the implementations that mimic potential mistakes by real programmers, and used the tool to identify executions that exhibited the resulting incorrect flow of information. The results are given in Fig. 2.

FlumeWiki. FlumeWiki [3] is a Wiki based on the MoinMoin Wiki engine [15], but redesigned and implemented using the Flume API to enforce desired DIFC properties. A simplification of the design architecture for FlumeWiki serves as the basis for the running example in Fig. 1. We focused on verifying the following properties:

– **Security:** Information from one `Worker` process should never reach another `Worker` process. Formally, $\mathsf{NoFlowHistory}(\mathsf{Worker}, \mathsf{Worker}, \emptyset)$.
– **Functionality:** A `Worker` process should always be able to send data to the `Handler` process. Formally, $\mathsf{DefiniteSingleStepFlow}(\mathsf{Worker}, \mathsf{Handler})$.

We created a buggy version ("Interference") by retaining the DIFC code that allocates a new tag for each process, but removing the code that initializes each new process with the tag. The results for both versions are presented in Fig. 2.

Apache. The Apache [7] web server modularizes implementations of policies for servicing requests. We analyzed the *preforking* module, which pre-emptively launches a set of worker processes, each with its own channel for receiving requests. We checked this module against the properties checked for FlumeWiki above. Because there was no preexisting Flume code for Apache, we wrote label-manipulation code by hand and then verified it automatically using our tool.

ClamAV. ClamAV [8] 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. We verified flow properties over the module that

ClamAV uses to scan files marked as sensitive by a user. Our results demonstrate that we are able to express and check a policy, *export protection* (given below as the security property), that is significantly different from the policy checked for the server models above. The checked properties are as follows:

– **Security:** ClamAV should never be able to send private information out over the network. Formally, $\mathsf{NoFlowHistory}(\mathsf{Private}, \mathsf{Network}, \emptyset)$.
– **Functionality:** ClamAV should always be able to read data from private files. Formally, $\mathsf{DefiniteSingleStepFlow}(\mathsf{Private}, \mathsf{ClamAV})$.

Because there was no DIFC manipulation code in ClamAV, we implemented a "manager" module that initializes private files and ClamAV with DIFC labels, similar to the scenario described in [2]. We introduced a functionality bug ("NoRead") into the manager in which we did not initialize ClamAV with the tags needed to be able to read data from private files. We introduced a security bug ("Export") in which the handler accidentally gives ClamAV sufficient capabilities to export private data over the network.

OpenVPN. OpenVPN [9] is an open-source VPN client. As described in [2], because VPNs act as a bridge between networks on both sides of a firewall, they represent a serious security risk. Similar to ClamAV, OpenVPN is a program that manipulates sensitive data using a bounded number of processes. We checked OpenVPN against the following flow properties:

– **Security:** Information from a private network should never be able to reach an outside network unless it passes through OpenVPN. Conversely, data from the outside network should never reach the private network without going through OpenVPN. Formally, $\mathsf{NoFlowHistory}(\mathsf{Private}, \mathsf{Outside}, \mathsf{OpenVPN}) \wedge \mathsf{NoFlowHistory}(\mathsf{Outside}, \mathsf{Private}, \mathsf{OpenVPN})$.
– **Functionality:** OpenVPN should always be able to access data from both networks. Formally, $\mathsf{DefiniteSingleStepFlow}(\mathsf{Private}, \mathsf{OpenVPN}) \wedge \mathsf{DefiniteSingleStepFlow}(\mathsf{Outside}, \mathsf{OpenVPN})$.

Because there was no DIFC manipulation code in OpenVPN, we implemented a "manager" module that initializes the networks and OpenVPN with suitable labels and capabilities. We introduced a bug ("NoRead") in which the manager does not initialize OpenVPN with sufficient capabilities to read data from the networks. We introduced another bug ("Leak") in which the manager initializes the network sources with tag settings that allow some application other than OpenVPN to pass data from one network to the other. Our results indicate that the approach allows us to analyze properties over bounded processes for large-scale programs.

The analyzes of FlumeWiki and Apache take significantly longer than those of the other modules. We hypothesize that this is due to the fact that both of these modules may execute using unbounded sets of processes and tags, whereas the other modules do not. Their abstract models can thus frequently generate non-deterministic values, leading to the examination of many control-flow paths.

Limitations. Although the tool succeeded in proving or finding counterexamples for all properties that we specified, we do not claim that the tool can be applied successfully to all DIFC properties. For instance, our current methods cannot verify certain correctness properties for the full implementation of FlumeWiki [3], which maintains a database that relates users to their DIFC state, and checks and updates the database with each user action, because to do so would require an accurate model of the database. The extension of our formalism and implementation to handle such properties is left for future work.

6 Related Work

Much work has been done in developing interprocess information-flow systems, including the systems Asbestos [16], Hi-Star [2], and Flume [3]. While the mechanisms of these systems differ, they all provide powerful low-level mechanisms based on comparison over a partially ordered set of labels, with the goal of implementing interprocess data secrecy and integrity. Our approach can be viewed as a tool to provide application developers with assurance that code written for these systems adheres to a high-level security policy.

Logical structures have been used previously to model and analyze programs to check invariants, including heap properties [12] and safety properties of concurrent programs [17]. In this paper, we used the semantic machinery of first-order logic to justify the use of random isolation, which was introduced in [6] to check atomic-set serializability problems.

There has been previous work on static verification of information-flow systems. Multiple systems [18, 19] have been proposed for reasoning about finite domains of security classes at the level of variables. These systems analyze information flow at a granularity that does not match that enforced by interprocess DIFC systems, and they do not aim to reason about concurrent processes.

The papers that are most closely related to our work are by Chaudhuri et al. [10] and Krohn and Turner [11]. The EON system of Chaudhuri et al. analyzes secrecy and integrity-control systems by modeling them in an expressive but decidable extension of Datalog and translating questions about the presence of an attack into a query. Although the authors analyze a model of an Asbestos web server, there is no discussion of how the model is extracted. Krohn and Turner [11] analyze the Flume system itself and formally prove a property of non-interference. In contrast, our approach focuses on automatically extracting and checking models of applications written for Flume and using abstraction and model checking. Our work concerns verifying a different portion of the system stack and can be viewed as directly complementing the analysis of Flume described in [11].

Guttman *et al.* [20] present a systematic way based on model checking to determine the information-flow security properties of systems running Security-Enhanced Linux. The goal of these researchers was to verify the policy. Our work reasons at the code level whether an application satisfies its security goal. Zhang *et al.* [21] describe an approach to the verification of LSM authorization-hook placement using CQUAL, a type-based static-analysis tool.

References

1. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R.: Labels and event processes in the Asbestos operating system. SIGOPS Oper. Syst. Rev. **39**(5) (2005) 17–30
2. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: OSDI. (2006)
3. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard os abstractions. In: SOSP. (2007)
4. Conover, M.: Analysis of the Windows Vista Security Model. Technical report, Symantec Corporation (2008)
5. Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., Yorav, K.: Efficient verification of sequential and concurrent C programs. Form. Methods Syst. Des. **25**(2-3) (2004) 129–166
6. Kidd, N.A., Reps, T.W., Dolby, J., Vaziri, M.: Finding concurrency-related bugs using random isolation. In: VMCAI. (2009)
7. Apache: Apache. http://www.apache.org
8. ClamAV: ClamAV. http://www.clamav.net
9. OpenVPN: OpenVPN. http://www.openvpn.net
10. Chaudhuri, A., Naldurg, P., Rajamani, S.K., Ramalingam, G., Velaga, L.: EON: Modeling and analyzing dynamic access control systems with logic programs. In: CCS. (2008)
11. Krohn, M., Tromer, E.: Non-interference for a practical DIFC-based operating system. In: IEEE Symposium on Security and Privacy. (2009)
12. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298
13. Harris, W.R., Kidd, N.A., Chaki, S., Jha, S., Reps, T.: Verifying information flow control over unbounded processes. Technical Report UW-CS-TR-1655, Univ. of Wisc. (May 2009)
14. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs (2002)
15. MoinMoin: The MoinMoin wiki engine. http://moinmoin.wikiwikiweb.de (December 2006)
16. Vandebogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., Mazières, D.: Labels and Event Processes in the Asbestos Operating System. ACM Trans. Comput. Syst. **25**(4) (2007) 11
17. Yahav, E.: Verifying safety properties of concurrent java programs using 3-valued logic. In: POPL. (2001)
18. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7) (1977) 504–513
19. Myers, A.C.: JFlow: practical mostly-static information flow control. In: POPL. (1999)
20. Guttman, J.D., Herzog, A.L., Ramsdell, J.D., Skorupka, C.W.: Verifying Information-Flow Goals in Security-Enhanced Linux. Journal of Computer Security (2005)
21. Zhang, X., Edwards, A., Jaeger, T.: Using CQUAL for static analysis of authorization hook placement. In: USENIX Security Symposium. (2002) 33–48