Verifying Information Flow Over Unbounded Processes

William R. Harris*, Nicholas A. Kidd*, Sagar Chaki†,
Somesh Jha*, and Thomas Reps*‡
* *University of Wisconsin; Madison, WI; USA; {wrharris, kidd, jha, reps}@cs.wisc.edu*
† *Soft. Eng. Inst.; Carnegie Mellon University; Pittsburgh, PA; USA; chaki@sei.cmu.edu*
‡ *GrammaTech Inc.; Ithaca, NY; USA*

Abstract

Distributed Information Flow Control *(DIFC) systems enable programmers to express desired information-flow policies, and enforce the policies via a reference monitor that restricts interactions between system objects, such as processes and files. Current research on DIFC systems focuses on the reference-monitor implementation, and assumes that the application correctly enforces the desired information-flow policy. The focus of this paper is a semi-automatic technique to verify that the application does indeed enforce the (high-level) policy. We perform a source-to-source transformation on an input C program, and then ask if the transformed program satisfies an LTL formula expressing the desired policy. The transformation soundly abstracts programs with a potentially unbounded number of processes and communication channels. We implemented our approach and evaluated it on a set of real-world programs.*

1. Introduction

Distributed Information Flow Control (DIFC) systems [1]–[4] allow application programmers to specify their own information-flow policies, and then enforce the policy in the context of the entire operating system. To achieve this goal, they maintain a mapping from system 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 program's data distribution by collecting and discarding the tags. The DIFC runtime system acts as a reference monitor for all inter-process communication, deciding whether or not a requested data transfer is allowed based on the labels of system objects.

For example, consider the diagram of a web server handling sensitive information given in Fig. 1. 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. Suppose that the server wants to enforce a *non-interference policy* requiring that information pertaining to one request, and thus in one Worker process,

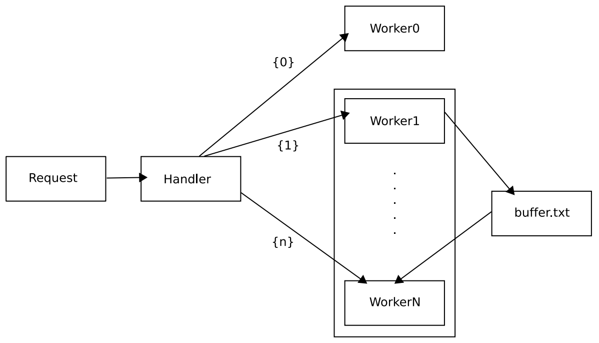

Figure 1. An inter-process diagram of a typical web-server

should never flow to another Worker process. This is a non-trivial policy to implement, especially because two corrupted processes acting in collusion could write and read data from a buffer not managed by the Handler. Nevertheless, careful use of DIFC mechanisms can ensure that the server adheres to the policy.

In addition to ensuring that the use of DIFC mechanisms enforce a desired security policy, the programmer must also ensure that the retrofitting of DIFC code to an existing system does not negatively impact the system's functionality. This is because DIFC mechanisms are able to block potentially any communication between objects. In the example, the Handler must be able to communicate with each Worker at all times. In addition, the server may want to allow each Worker to be able to create, write, and read from its own independent set of files. An overly restrictive policy implementation could disallow such behaviors. Our example thus hints at a fundamental tension between security and functionality in such systems. A naïve system that focuses solely on functionality allows information to flow between all entities. Conversely, a system could be made completely secure in a trivial way, but could cripple functionality.

Difficulties in reasoning about such properties are exacerbated by concurrency, because concurrency tremendously increases the complexity of manual reasoning. Our proposed (partial) solution to this problem is to leverage progress in model checkers for concurrent software [5]–[7] that check concurrent programs against temporal logic (e.g., linear temporal logic, or LTL) properties. However, the translation from arbitrary, multiprocess systems to systems that can be reasoned about by concurrent model checkers in itself poses two key non-trivial challenges:

1) The number of processes spawned, communication channels created, and label values used by the reference-monitor is unbounded. However, a model checker will only work on models that use bounded sets of these entities. Moreover, the models constructed from these abstractions must be sound approximations of the original system in the sense that if a security or functionality property holds for the model, then the property must hold for the original program.
2) Model checkers must address the *state-explosion problem* [8], where the state space of a system grows exponentially with the number of components. A naïve treatment of concurrency yields an enormous state space to be searched.

The contributions of this work are as follows:

1) We describe a language for specifying information-flow properties that can capture common properties desired for real-world systems.
2) We describe a method for abstracting programs that potentially create an unbounded number of processes into a model that consists of a finite number of processes. We use *random isolation* [7] to ensure that the model is a sound over-approximation. Furthermore, the abstracted models' state spaces are greatly reduced in size from those of a naïve translation.
3) We implemented the abstraction as a program translation in CIL [9]. The translator accepts a program written for the API of Flume [3], a DIFC system executed as a UNIX process that can be used to monitor information-flow for UNIX programs, and a desired information flow property. It translates the program into an abstracted model in C, and uses the concurrent C model checker Copper [5] to verify that the model—and thus the program—satisfies the desired property.
4) We applied this tool to check desired properties for several real-world programs. We semi-automatically extracted models of modules of Apache [10], FlumeWiki [3], ClamAV [11], and OpenVPN [12] instrumented with our own label-manipulation code, and verified the desired properties in times ranging from a few seconds to about 1.5 hours. The results given in §5 demonstrate that the high level of abstraction in inter-process information-flow systems, while coarse, allows us to construct reasonable, checkable models from real-world programs.

While there has been prior work [13], [14] in the application of formal methods for checking properties of actual DIFC systems, our work is unique in providing a semi-automatic method for checking that an *application* satisfies a proof of correctness under the rules of a given DIFC system. Combining our techniques with the recent verification of the Flume reference monitor implementation [14] defines the first end-to-end system that is able to verify that a program implements a high-level information-flow policy.

The rest of this paper is organized as follows: §2 describes existing work on which our abstraction builds. §3 gives an informal overview of our techniques. §4 gives the technical description. §5 describes our experimental evaluation. §6 discusses related work.

2. Preliminaries

We target the Flume [3] DIFC system; however, our abstraction techniques should work with little modification for most DIFC systems.

2.1. Flume

Flume is implemented as a reference monitor that runs as a user-level process, but uses a Linux Security Module [15] for IPC interposition. Thus, it monitors current Linux programs with few small modifications to the application code. We briefly discuss the Flume datatypes and API functions relevant for information-flow properties of interest, and direct the reader to [3] for a complete description.

- **Tags**. A tag is an atomic element created by the monitor. Applications request new tags from the monitor, and add and remove the ability for other processes to remove tags to control data distribution.
- **Labels**. A label is a set of tags.
- **Capabilities**. A capability is either a positive or negative attribute of an object that permits label manipulation: t^+ denotes that the object may add t to its label, and t^- denotes that it may remove t.
- **Tokens**. To monitor all IPC calls that send data in and out of a process, the monitor does not allow processes to create their own file descriptors. Rather, if a process wishes to create a pipe, it asks the monitor for a new *channel*, and the monitor returns a pair of *tokens*. A process may either *pass* tokens to a spawned process or *exchange* tokens for file descriptors from the monitor. After exchanging a token, the application treats the resulting descriptor as a standard descriptor (that may point to a file or one end of a pipe), but the descriptor in fact points to a channel monitored by Flume, as described below. Because a token can only be exchanged for a descriptor, this mechanism ensures that at most one process may ever own a descriptor. For generality, we refer to descriptors as *endpoints* in the sequel.

For each process, Flume maintains a secrecy label, an integrity label, and a capability set. For this paper, we restrict ourselves to a discussion of Flume's secrecy labels and do not model the global capability set. We leave the modeling of both as directions for future work.[1] A process has the ability to:

- Create a new tag t and add or remove t^+ and t^- from the global capability set. When a process creates a tag t, it *owns* t, t^+, and t^-.
- Manipulate label variables using set operations.
- Redefine its label. Let C^+ denote the positive capabilities of a process and let C^- denote the negative capabilities. The redefinition of a label from l to l' is allowed if and only if $l' - l \subseteq C^+$ and $l - l' \subseteq C^-$.
- Redefine the labels of endpoints that the process owns with the same restrictions that apply to its own label.
- Exchange a token to claim sole ownership of the corresponding endpoint.
- Send and receive data over channels. Suppose that a process p with label l_p attempts to send data over endpoint e with label l_e. Before allowing the send, the monitor ensures that $l_p \subseteq l_e$. On a receive from e, the monitor similarly ensures that $l_e \subseteq l_p$.
- Spawn a new process. Flume provides a single API call `spawn` that effectively combines the `fork` and `exec` UNIX system calls. It takes as input the filepath of a binary and starts a new process that executes the binary while continuing execution in the current process. The `spawn` function takes two additional arguments: the initial tokens, the initial label, and initial capability set of the new process. The label must be one that the process could legally define as its own within the above restrictions. The capability set must be a subset of the process's capabilities. Thus, processes are given tags in their label without the ability to add or remove the tag from their label.

Lastly, note that Flume interposes itself on accesses to the file system, labeling files and checking accesses to them in a manner directly analogous to channel accesses. Thus information leakage through the file system is prevented.

Returning to the example from Fig. 1, note the pseudocode in Fig. 2 that enforces non-interference between Worker processes. The Handler perpetually polls for a new HTTP request, and upon receiving one, it prepares a new Worker process for execution. To do so, it first has Flume create a new tag which it stores in a singleton label `lab`, and then has Flume create a new channel. The Handler then launches the Worker process, setting its initial secrecy label to `lab`, not giving it the capability to add or remove the tag in `lab`, 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 flowing from the new Worker unless the Worker lowers its label. However, the Worker cannot remove the tag from its label, so this is impossible.

```
int Handler() {
  Endpoint_set S;
  int data = 0;
  while (*) {
    Request r = get_next_http_request();
    for (int i = 0; i < 1000; i++)
      data = data * crypto_func(data);
    Label lab = create_tag();
    Token t0, t1;
    create_channel(&t0, &t1);
    data = recv(claim_token(t0));
    spawn("/usr/local/bin/Worker", { t1 }, lab, { }, r); } }
```

Figure 2. Flume pseudocode for how a server can enforce the same-origin policy. Code removed by slicing appears in frames.

The above example shows that tags and labels in DIFC systems allow programmers to control the information flow of their multi-process programs. Our goal is to leverage concurrent model checking to automatically prove that these tag and label manipulations implement the desired security policy of isolating the information in each Worker process.

2.2. Random Isolation

Random isolation [7] is a technique for establishing properties about the behavior of an unbounded set of indistinguishable objects. The technique randomly selects one object from the set, and isolates it by attaching to it a distinguishable name. A program analysis then treats the randomly-isolated object specially, i.e., as a non-summary object or a singleton set, which allows the analysis to reason more precisely about the behavior of the randomly-isolated object. Finally, because it is chosen randomly from a set of indistinguishable objects, a proof that a property holds for the randomly-isolated object applies to *each* object in the set.

Our technique applies random isolation to soundly model the potentially unbounded sets of processes, channels, and tags. We refer to a distinguished object as a *non-summary object* (e.g., a non-summary process), and model the rest of the objects with a *summary object* (e.g., one summary process models all non-distinguishable processes).

1. Modeling integrity in Flume is a dual of problem of modeling secrecy. Labels are checked using subset comparison but the "opposite direction."

2.3. Predicate Abstraction-based Model Checking

In §4.3, we discuss techniques for greatly reducing the state-space of individual processes and approximating the effects of an unbounded number of processes with a finite process with special semantics. Even in the presence of these techniques, the model-extraction phase typically constructs models with state spaces on the order of 10^{25}. Thus software model checking (SMC) is applied to make the verification of such state spaces feasible. The key idea behind SMC is to combine *predicate abstraction* [16] and counterexample-guided abstraction-refinement [17] to iteratively, and automatically, create models that are precise enough to establish the properties of interest. In particular, this approach was implemented on top of the Copper [2] SMC tool.

3. Overview

We now give an informal description of the reduction from arbitrary multi-process systems to finite models. Recall from §1 and §2.1 that the DIFC system maintains a label for every process, file endpoint, and file. This implies an abstraction at the level of system objects. In other words, a DIFC system assumes that any data read by a process p_r from a process p_s is reflected in the entire memory space of p_r. This contrasts with JFlow [18], in which information flow is tracked at the level of variables. Because information is assumed to flow to all variables, we adopt an abstraction that merges almost all program variables. In other words, we replace the set of all process variables that do not denote process spawn points or channel endpoints with a single *taint label* that tracks the information reaching the process. Definitions of the original variables are ignored. If such variables occur in the predicates that guard flow to an IPC call, then the guard expression is replaced with a non-deterministic value. After performing this abstraction, we are left with simplified programs that perform only the following significant operations:

1) Processes create channels.
2) Processes send and receive data over channels.
3) Processes perform set operations to manipulate labels. These operations include generating new tags with new capabilities and adding them to labels.
4) Processes launch other processes.

This implies that there are three key program entities that we must model to some degree of accuracy, yet are unbounded: the set of executing processes, the set of possible tags and capabilities, and the set of channels.

3.1. Abstracting processes

For a given execution of a system, let G be the set of all processes that begin execution from the same program point. Abstraction must soundly approximate the effect of all processes in G in the original program with a bounded set G' of processes that execute in the model program. To do so, the abstraction lets G' consist of two processes. One of the processes, the non-summary process p_{non}, behaves exactly like one of the original processes in G. This process is assumed to execute under standard, *non-summary* program semantics. However, the other process in G', denoted as p_{sum}, executes under a *summary semantics*. The summary semantics ensures that if there is a possible execution of the system that generates a sequence s of sends and receives, then p_{sum} has an execution that generates a supersequence of s. §4 contains a proof that this property is sufficient to ensure that if the system consisting of the processes in G' is free of any information-flow violations, then so too is the system consisting of the processes in G.

To implement the summary semantics, we alter the implementation of `spawn` to non-deterministically launch a summary process or at most one non-summary process. Summary semantics differs from non-summary semantics in two ways:

1) Under non-summary semantics, when a process is spawned, the parent process passes it an initial set of tokens, an initial label, and an initial set of capabilities. In summary semantics, the first call to `spawn` that launches the summary process launches it with the argument token set, label, and capabilities. However, each subsequent call, instead of launching a new process, merges its token, label, and capability arguments into conservative approximations of the initial token and labels. For the target information-flow properties, it is sufficient to maintain an over-approximating set of token values. However, to check soundly for security and functionality constraints, a pair of labels that under- and over-approximate the possible initial label, along with a pair of capability sets that under- and over-approximate the capabilities of the summary process, must be maintained.
2) Rather then executing the process code once, the summary process continuously iterates over its code segment. Each time that it completes execution of the code segment, the system re-initializes its token, label, and capability variables non-deterministically to values contained in the conservative approximation.

These summary semantics can be modeled in a model C program, allowing a model checker to verify properties without changes to its internal algorithm.

For the example in Fig. 1, let G be the unbounded set of Worker processes. The abstraction set G' contains one non-summary process p_{non}, depicted in Fig. 1 as Worker_0; and then summarizes the remaining processes with a single summary process p_{sum}, depicted with a single frame containing the remaining Worker processes.

2. `http://www.sei.cmu.edu/pacc/copper.html`

3.2. Abstracting tags

Similar to the abstraction of processes, the set of allocated tags is abstracted to one non-summary tag and one summary tag with a non-summary semantics for tag manipulation. The summary tag changes tag and label manipulation in two ways:

1) When the model calls `create_tag`, the function non-deterministically returns the summary tag t_{sum} or the non-summary tag t_{non}. The latter is done at most once.
2) Suppose that we wish to check if a label l is a subset of a label m. This is similar to standard subset comparison except in the case where l contains the summary tag and is a subset of m. To see this, note that the summary tag t_{sum} is allocated in place of multiple concrete tags. Thus, its occurrence in l may summarize an entirely different set of concrete tags in l than the occurrence in m. Thus, while the normal subset comparison employed for label checking maps to Boolean values, the subset operation over non-summary and summary tags maps to a three-valued domain and in the case given above, evaluates to "unknown." In the model, subset comparison returns a non-deterministic value in the above case.

Returning to the example in Fig. 1, whenever `Handler` invokes `create_tag`, it may return the summary tag t_{sum}. Furthermore, on at most one invocation it may return the non-summary tag t_{non}.

3.3. Abstracting channels

Finally, the set of channels is abstracted to a bounded set containing a singleton channel and a summary channel. The summary channel affects channel manipulation in the following ways:

1) When the model code invokes `create_channel`, the function non-deterministically returns the summary channel c_{sum} or it returns the non-summary channel c_{non}. It will return the non-summary channel at most once.
2) If a process updates the labels of an endpoint of a summary channel, we perform a weak update of the label value and maintain an under-approximation and over-approximation of the label value.
3) Suppose that a process wishes to send data over endpoint e of the summary channel. To determine conservatively if the send may succeed, we check the label of the process against the over-approximation of the endpoint label. Similarly, to determine conservatively if the send may fail, we check the label of the process against the under-approximation of the endpoint label.

Consider again the example in Fig. 1. Whenever `Handler` invokes `create_channel`, `create_channel` may return the endpoints of the summary channel $(e^0_{\text{sum}}, e^1_{\text{sum}})$. Furthermore, on at most one invocation it may return $(e^0_{\text{non}}, e^1_{\text{non}})$, the endpoints of the non-summary channel.

3.4. Potential Loss of Precision

We now analyze the effect of all of these techniques in concert to see how the abstraction handles the running example. In Fig. 1, we wish to prove that the Handler configures and launches each Worker process in such a way that information from no Worker process can reach a different Worker process. The model program produced from abstraction spawns at most one singleton Worker process and a single summary Worker process. The model program creates at most one singleton tag and one summary tag, and similarly for channels. When a model checker checks the model program, it attempts to find a path of execution that allows one process to send data to the other. One such execution of the model that it finds proceeds essentially as follows:

1) The Handler asks for a tag, receives t_{sum}, and launches the non-summary process with this as an initial label.
2) The Handler asks for a tag, receives t_{sum} again, and launches the summary process with initial tag t_{sum}.
3) The non-summary process opens a file on the filesystem modeled as a channel and writes data to it.
4) The summary process opens the same channel and reads data from it.

The model checker thus concludes that a violation is possible. However, the original code correctly marked each Worker with a unique tag that each could not remove, enforcing isolation between Workers. The counterexample is thus spurious.

3.5. Choosing Randomly-Isolated Objects

Without compensating for the loss of precision introduced by abstraction, the tool may not be able to verify that the desired flow policy has been implemented correctly. The key insight behind our approach is that we are able to effectively choose which objects are randomly isolated, and thus ignore many spurious counterexamples—especially those involving only summary processes like that described above.

Random isolation is a technique for adding one bit of instrumentation for each type of object, where the bit is only set for one object of each object type (e.g., only one process can have the instrumentation bit set). Consider a concrete execution trace π of the program such that no instrumentation bits are set for any object type, and π violates the flow policy. An equivalence class Π of traces is defined by π, where the only distinguishing characteristic between two traces π_1 and π_2 in Π is which objects are randomly isolated. To verify that the program adheres to the flow policy, it is sufficient to consider only one representative trace $\overline{\pi} \in \Pi$ because every trace in Π has the same concrete behavior (the only difference being which objects have the instrumentation bit set). Thus, when model checking the program, for each equivalence class of traces, the trick is to choose the trace $\overline{\pi}$ that is most "advantageous" from the standpoint of retaining enough precision to establish or refute the property of interest.

Because the properties of interest are always of the form, "Can information flow from a process of type G to a process of type H?", it is desirable to choose the representative trace $\overline{\pi}$ such that if $\overline{\pi}$ violates the flow policy, then in the abstraction of $\overline{\pi}$, the illegal flow originated from a *sufficiently-concrete process*. A sufficiently-concrete process is one whose *initial* token set, label, and capability set are not solely summary objects (e.g., the (abstract) label cannot be the set $\{t_{\mathsf{sum}}\}$). This is accomplished by inserting instrumentation predicates in the model so that the model checker only considers (abstractions of) concrete execution traces where the property is violated and flow began from a sufficiently-concrete process.

Returning to the example in Fig. 1, one can see how the model checker can verify that the model does not violate the desired flow property:

1) For a violation to occur, the model checker only considers executions in which the Handler asks for and receives the non-summary tag t_{non}, and launches a non-summary process with t_{non} in its initial label.
2) Due to the program abstraction, all other Worker processes are modeled by a summary process whose labels consist of the summary tag.
3) Because no other Worker label contains t_{non}, an illegal flow is impossible.

4. Program Abstraction

We now give a formal definition of our program-and-specification transformation. The overall goal is to take an original program written for the Flume API with minimal annotations, along with a high-level specification of correctness, and produce either a proof that the program adheres to the specification, or a (potentially spurious) execution trace demonstrating that it does not.

4.1. Inputs

4.1.1. Subject programs. For discussion purposes, we assume that the analysis works over a simple imperative language with assignments, function calls, and if and while statements for control flow. Expressions are constructed from integers, symbols denoting process-entry points, and the types T_{Flume} provided by the Flume API: tags, labels, capabilities, tokens and endpoints. For expressions with Flume types, only equality comparisons are allowed, and the standard binary operations are allowed over integers.

The semantics of our language is standard with a few exceptions for modeling Flume semantics [3]:

- Consider the call statement `spawn(f, T, L, C)`. Here, f is a function in the code, T is a set of tokens, L is a label, and C is a set of capabilities. The call to `spawn` causes a fresh process to begin execution in the function f with an argument storing the initial set of channel tokens T. The model maintains a mapping label, which maps each process and endpoint to its label. The call to `spawn` updates label with $\mathsf{label} := \mathsf{label}[p \mapsto L]$. Similarly, the model maintains a mapping cap from each process to its capability. The call to `spawn` updates the mapping with $\mathsf{cap} := \mathsf{cap}[p \mapsto C]$. The process's *input configuration* is the triple (T, L, C).
- Consider the assignment statement `(t, t') = create_chan()`. This yields two tokens. Throughout execution, Flume maintains a map endp from each token to the endpoint for which it may be claimed.
- Consider the statement `e = claim(t)` called by a process p. This returns the endpoint $\mathsf{endp}(t)$. The model also maintains a mapping owner from each endpoint to the process that owns it. This assignment updates the owner with $\mathsf{owner} := \mathsf{owner}[\mathsf{endp}(t) \mapsto \{p\}]$. For a non-summary endpoint e_{non}, the set $\mathsf{owner}(e)$ is a singleton set, but this restriction does not apply for the summary endpoints as defined later. The owner mapping is lifted to channels with $\mathsf{owner}(\mathsf{chan}(e^0, e^1)) = \mathsf{owner}(e^0) \cup \mathsf{owner}(e^1)$.
- Suppose that a process p calls $\mathsf{set\_proc\_label}(L)$. Let C_p^+ and C_p^- denote the capabilities of p to add and remove tags, respectively. The model then checks that $L - \mathsf{label}(p) \subseteq C_p^+$ and $\mathsf{label}(p) - L \subseteq C_p^-$. If both of these relations hold, then it performs the update $\mathsf{label} := \mathsf{label}[p \mapsto L]$.
- If a process p calls $\mathsf{set\_endp\_label}(e, L)$, then the model performs the update $\mathsf{label} := \mathsf{label}[e \mapsto L]$.
- Calls to `send` and `recv` directly affect the trace semantics defined in §4.2.

Let F_{Flume} be the set of functions defined by Flume (`create_tag`, `create_chan`, etc.).

4.1.2. Specification. Tags are modeled as a set T of atomic elements and labels are modeled as elements in 2^T. A non-summary endpoint is denoted by $e_{\mathsf{non}}(i, L)$, where $i \in \{0, 1\}$ denotes which "end" of the channel the endpoint refers to, and L is a label. A channel is modeled as a term

3. The updates that bind summary processes to values are strong, but sound. The possible initial states of the summary process are updated weakly, ensuring soundness.

$\mathsf{chan}(e^0, e^1)$ constructed directly from its endpoints. Each endpoint belongs to exactly one channel; let $\mathsf{chan}(e)$ denote the channel to which an endpoint e belongs.

Let a specification be a (possibly empty) list of security properties along with a (possibly empty) list of functionality properties. Let a non-summary process be a term $\mathsf{proc}(f, i)$ where f is the entry point of the process in the code and i is a unique index among processes with the same entry point. Let a *process group* be any set of processes that have the same entry point (and thus unique indices) and denote the common entry point of a group G as $\mathsf{entry}(G)$. Let $\mathcal{G}$ denote the set of all process groups. A *security property* is of the form $G \not\to^* H$ where G, H are process groups. This property implies that information from a given process in G must never reach a process in H. The property $G \not\to^* G$ means that information from a process in G does not reach a *different* process in G. Such a property where $G = \mathsf{Worker}$ encodes the same-origin policy from Fig. 1.

For process groups G, H, let a *functionality property* be denoted as $G \to H$ and interpret it to mean that whenever a process in G attempts to send data through endpoint e^0 over channel $\mathsf{chan}(e^0, e^1)$, where $\mathsf{owner}(e^1) \in H$, then the send must be successful.

An information flow property is either a security property or a functionality property.

4.2. Trace Semantics

We now define our program abstractions, and prove that they are sound with respect to our information-flow properties. At a high level, the structure of the proof is as follows:

1) The *trace* of an execution of a program P is the sequence of actions it performs that are relevant to the flow properties.
2) We define a set of renaming functions that transform a trace defined over one set of processes and channels, into a trace defined over a different set with certain restrictions. If an original trace violates some property, then there exists at least one renaming function that produces a trace that also violates the property.
3) We define an abstracted program $\mathfrak{A}(P)$ that executes over a finite set of system objects.
4) We prove that for each trace from P renamed to be over objects in $\mathfrak{A}(P)$, there exists one trace in $\mathfrak{A}(P)$ that contains it as a subsequence. From this it follows that $\mathfrak{A}(P)$ soundly approximates P.

Whether or not a program satisfies an information flow property is determined by the interprocess `send`s and `recv`s executed by the program. Let both of these events be actions. A program generates a send action $\mathsf{send}(p, c)$ when process instance p successfully executes the call `send(e)`, where $\mathsf{chan}(e) = c$, and generates an action $\mathsf{blocked\_send}(p, c)$ when the call is blocked by Flume. Symmetrically, a receive action $\mathsf{recv}(c, p)$, is generated when a process instance p successfully calls `recv(e)`, where $\mathsf{chan}(e) = c$, and an action $\mathsf{blocked\_recv}(c, p)$ is generated when Flume blocks the call. A *trace* is a sequence of actions. If a trace σ is a subsequence of a trace τ, this is denoted by $\sigma \preceq \tau$. The number of actions in a trace σ is denoted by $|\sigma|$.

Definition 1 (Transfer). *A trace σ* transfers *from process group G to group H if there exists $\tau \preceq \sigma$ of the form:*

$$(\mathsf{send}(p_0, c_0), \mathsf{recv}(c_0, p_1), \mathsf{send}(p_1, c_1), \ldots, \mathsf{recv}(c_{\frac{|\tau|}{2}-1}, p_{\frac{|\tau|}{2}}))$$

where the following holds for τ:

$$p_0 \in G \wedge p_{\frac{|\tau|}{2}} \in H \wedge p_0 \neq p_{\frac{|\tau|}{2}}$$

This is denoted with the relation $\mathsf{transfers}(G, H, \sigma)$.

Definition 2 (Blocking). *A trace σ* blocks *from a process group G to process group H if there exists $\tau \preceq \sigma$ such that*

$$\begin{aligned}(\tau = &(\mathsf{blocked\_send}(p_0, c_0)) \wedge p_0 \in G \\ &\wedge \mathsf{owner}(c_0) \cap H \neq \emptyset) \\ \vee\ (\tau = &(\mathsf{blocked\_recv}(c_1, p_1)) \wedge p_1 \in H \\ &\wedge \mathsf{owner}(c_1) \cap G \neq \emptyset)\end{aligned}$$

This is denoted by $\mathsf{blocks}(G, H, \sigma)$.

Satisfaction of formulas is now defined for traces and programs.

Definition 3 (Trace satisfaction). *A trace σ* violates *a security property $\varphi = G \not\to^* H$ if and only if* $\mathsf{transfers}(G, H, \sigma)$. *Trace σ* violates *a functionality property $\varphi = G \to H$ iff* $\mathsf{blocks}(G, H, \sigma)$. *Trace σ satisfies φ, denoted as $\sigma \models \varphi$, if it does not violate φ. Trace violation is denoted by $\sigma \not\models \varphi$.*

Definition 4 (Program satisfaction). *Let P be a program that can generate a set of traces $\mathsf{Tr}(P)$ and let φ be a flow property. Program P satisfies φ, denoted by $P \models \varphi$, if for all $\sigma \in \mathsf{Tr}(P)$, it is the case that $\sigma \models \varphi$. The program P violates φ, denoted as $P \not\models \varphi$, if it does not satisfy φ.*

The program abstractions restructure programs that may operate over unbounded sets of system objects to operate over bounded sets of objects. We thus need a notion of translating a trace over one set of system objects into a trace over a different set of objects and for defining when such a translation preserves sufficient structure in the new trace. This notion is captured by defining *renaming functions* over system objects, actions, and finally traces. The following condition helps capture the structure expected of such a translation.

Definition 5 (One-to-one modulo U). *Let $f : S \to T \cup U$ be a function where $T \cap U = \emptyset$. f is* one-to-one modulo U *if f restricted to all elements that map to T is one-to-one.*

Well-formed renaming functions over system objects are now defined. First, let T_1, T_2 be sets of tags. A function $h_t : T_1 \rightarrow T_2$ is a well-formed tag-renaming function if there exists some $t_s \in T_2$ such that h_t is one-to-one modulo $\{t_s\}$. Intuitively, t_s corresponds to the summary tag. Lift a tag-renaming function h_t to a label-renaming function h_l with $h_l(l) = \{h_t(t) \mid t \in l\}$. Similarly, let C_1, C_2 be sets of channels. A function $h_c : C_1 \rightarrow C_2$ is a well-formed channel-renaming function if there exists some $c_s \in C_2$ such that h_c is one-to-one modulo $\{c_s\}$. Intuitively, c_s corresponds to the summary channel. Finally, for each process group of size n, let there be some element $i \in \mathbb{Z}_m$, $m \leq n$, such that there exists a function $h_n : \mathbb{Z}_n \rightarrow \mathbb{Z}_m$ that is one-to-one modulo $\{i\}$. Intuitively, i corresponds to the index of the summary process of a group. A well-formed renaming function over sets of process instances P_1, P_2 is then any function $h_p : P_1 \rightarrow P_2$ of the form $h_p(\mathsf{proc}(F, i)) = \mathsf{proc}(F, h_n(i))$.

The renaming functions for renaming tags, channels, and processes can be lifted to functions for renaming actions and traces. Observe that an alphabet of actions can be constructed directly from a set of channels and a set of process instances. For an action alphabet A, this is denoted by $A = (C, P)$. Let A_1, A_2 be action alphabets. A function $h_a : A_1 \rightarrow A_2$ is a well-formed action-renaming function if there exists a well-formed channel-renaming function h_c and a well-formed process-renaming function h_p such that if $a = \mathsf{send}(p, c)$, then $h_a(a) = \mathsf{send}(h_p(p), h_c(c))$ and similarly for the $\mathsf{recv}, \mathsf{block\_send}, \mathsf{block\_recv}$ actions. Finally, let A_1, A_2 be action alphabets. A function h_s is a well-formed trace-renaming function from traces over A_1 to traces over A_2 if there exists a well-formed action renaming function $h_a : A_1 \rightarrow A_2$ such that $h_s(a_0, a_1, \ldots a_n) = (h_a(a_0), h_a(a_1), \ldots, h_a(a_n))$.

Renaming functions preserve relevant properties of traces:

Lemma 1 (Trace preservation). *Let σ be a trace over an action alphabet $A = (C, P)$ where the process instances are partitioned into groups $G_0, G_1, \ldots, G_n$. Let $A' = (C', P')$ be a distinct action language with process instances partitioned into groups $G'_0, G'_1, \ldots, G'_n$ such that $\mathsf{entry}(G_i) = \mathsf{entry}(G'_i)$ and $|G'_i| = min(|G_i|, 2)$. If there exists some flow property φ such that $\sigma \not\models \varphi$, there exists some $h_s : S \rightarrow S'$ such that $h_s(\sigma) \not\models \varphi'$ where φ' is φ replaced with the corresponding process groups.*

Proof: See the Appendix. □

Furthermore, property violation is preserved across subsequence containment.

Lemma 2 (Subsequence preservation). *Let $\sigma \not\models \varphi$ and let $\sigma \preceq \sigma'$. Then it is the case that $\sigma' \not\models \varphi$.*

Proof: Suppose that σ contains some subsequence τ that violates a flow property. It is immediate that σ' contains τ as a subsequence, thus violating the property. □

4.3. Code Abstraction

The code-abstraction translation is based on the insight that because DIFC systems assume that all variables in the memory space of a process contain the same information, aggressive but accurate abstractions can be performed by removing all variables that do not directly affect the flow of information across system objects. To perform this abstraction, let an *IPC instruction* be any statement of the form $x := f(\ldots)$ where $f \in F_{Flume}$. An *IPC statement* is defined recursively to be any of the following:

- An IPC instruction.
- A control-flow statement containing an IPC statement.
- A call to a function whose body contains at least one IPC statement.

Given a program P, code abstraction yields a program P' that contains only the IPC statements of P and only functions that contain at least one IPC statement. A statement that defines a non-Flume variable is removed entirely. If the guard of a control-flow statement contains any variable of non-Flume type, then the entire guard is replaced with code that produces a fresh non-deterministic value each time that the point in the code is reached.

Note that while Flume is implemented as an API for C programs, the target language is assumed not to contain pointers. We treat the analysis of pointers independently and assume that transformations similar to those performed in [19] have been performed, translating pointer accesses to conditional control-flow structures over integers.

The code abstraction described above is sound with respect to the information-flow properties.

Lemma 3 (Soundness of code abstraction). *Let P be an original program and let $\mathfrak{C}(\cdot)$ perform the code abstraction defined in §4.3. Then for any flow property φ, if $\mathfrak{C}(P) \models \varphi$ then $P \models \varphi$.*

Proof: See the Appendix. □

4.4. Finite Object Abstraction

To allow a concurrent model checker to verify properties about the executions of a model, the model must operate over a finite set of system objects. After performing code abstraction, there remain three unbounded system objects left to abstract: tags (and thus labels and capabilities), channels, and executing processes.

4.4.1. Finite tag abstraction. We now define a program transformation that transforms programs to execute over a finite set of tags. For a model to describe soundly programs that may generate an unbounded number of tags, the outcome of any comparison based on the state of tags in the original program must be approximated in the new model. To achieve this, the model of the Flume API function `create_tag` checks if it already has allocated the non-summary tag. If not, it yields the summary tag and if so, it non-deterministically yields the summary tag or the non-summary tag. Let this new version of `create_tag` be named $\texttt{create\_tag}_\alpha$. Equipped with such a function, programs may create labels that are collections of the non-summary tag and the summary tag. It is easy to construct examples that demonstrate that the application of the standard set-difference ("$-$") or subset comparison ("$\subseteq$") operators over such labels in the model program would lead to an unsound approximation of the original program.

The standard label-difference operator "$-$" can be extended to one that operates over such labels to yield an approximation. It is defined as:

$$l -_\alpha m = (l - m) \cup (l \cap \{t_s\})$$

The operator $\subseteq$ is then extended to $\subseteq_\alpha$, a subset operator over labels that may contain abstract tags:

$$l \subseteq_\alpha m = \begin{cases} \{1\} & \text{if } l \subseteq m \wedge t_s \notin l \\ \{0\} & \text{if } l \not\subseteq m \\ \{0,1\} & \text{otherwise} \end{cases}$$

To support this abstraction, we assume that the model checker has the ability to represent a set of boolean values and test the set in a manner that conservatively approximates the set of behaviors possible. In Copper, the third case of $\subseteq_\alpha$ is implemented as a non-deterministic value. Copper then considers executions in which the non-deterministic value is 0 as well as executions for which the value is 1.

4.4.2. Finite channel abstraction. Similar to the finite tag abstraction, we now define a program transformation that abstracts a program to operate over a finite set of channels. The function `create_chan` is translated to a new function $\texttt{create\_chan}_\alpha$ that non-deterministically returns the pair of summary-channel endpoints or returns a fresh pair of endpoints for the non-summary channel at most once. The model of non-summary endpoints is given in §4.1.2. A summary endpoint is modeled as a term $e_{\mathsf{sum}}(i, U, O)$ where i is defined as in the non-summary case, and U, O are labels that simultaneously under- and over-approximate the label of an endpoint. When a process p claims the endpoint, labels U and O are initialized with $U = O = \mathsf{label}(p)$. The summary channel is then a channel constructed as $\mathsf{chan}(e^0_{\mathsf{sum}}, e^1_{\mathsf{sum}})$ from the two summary endpoints. Flume operations that depend on endpoints now have the following semantics:

- `claim_token(t)`. Let this function be called by a process p. For the non-summary token t_{non}, the model updates the owners mapping as $\mathsf{owners} := \mathsf{owners}[\mathsf{endp}(t) \mapsto \{p\}]$ and returns the endpoint $\mathsf{endp}(t)$. For the summary token t_{sum}, the model performs a weak update of the set of owners of $\mathsf{endp}(t)$ denoted as: $\mathsf{owners} := \mathsf{owners}[\mathsf{endp}(t) \mapsto \mathsf{owners}(\mathsf{endp}(t)) \cup \{p\}]$. It then returns the endpoint $\mathsf{endp}(t)$.
- `set_endp_label(e, L')`. If $e = e_{\mathsf{non}}(i, L)$ is a non-summary endpoint, then this transitions e from $e_{\mathsf{non}}(i, L) \Rightarrow e_{\mathsf{non}}(i, L')$. If it is a summary endpoint, then the effect is $e_{\mathsf{sum}}(i, U, L) \Rightarrow e_{\mathsf{sum}}(i, U \cap L', O \cup L')$.
- `send(e)`. Let `send` be called by a process p. Suppose that $e = e_{\mathsf{non}}(i, l)$ is a non-summary endpoint. If $1 \in (\mathsf{label}(p) \subseteq_\alpha \mathsf{label}(e))$, then this call generates the action $\mathsf{send}(p, c)$. If $0 \in (\mathsf{label}(p) \subseteq_\alpha \mathsf{label}(e))$, then the call generates the action $\mathsf{send\_blocked}(p, c)$. If $(\mathsf{label}(p) \subseteq_\alpha \mathsf{label}(e)) = \{0, 1\}$, then the call non-deterministically generates both actions. Now suppose that $e = e_{\mathsf{sum}}(c, i, U, O)$ is a summary endpoint. The set of guard Boolean values $G = \bigcup_{L \in U, O} (\mathsf{label}(p) \subseteq_\alpha L)$ approximates whether or not Flume will allow the communication. [4] Actions are then generated based on the values in G in the same manner as the values of $\subseteq_\alpha$ are checked in the non-summary case.
- `recv(e)`. Let `recv` be called by process p. The generation of recv and block_recv actions from this call is entirely symmetric to the case of `send`. The only significant difference is that when receiving data from a summary endpoint $e_{\mathsf{sum}}(e, U, O)$, the set that over-approximates the success of `recv` is: $\bigcup_{L \in U, O} L \subseteq_\alpha \mathit{label}(p)$.

4.4.3. Finite process abstraction. Finally, we define an abstraction that transforms a program P into a program P' that executes over a bounded number of processes. To do so, for a process p we construct a summary process p_{sum} such that if P launches an unbounded set of instances of p, then P' launches p_{sum} in place of the unbounded set. The summary semantics of p_{sum} maintain that P' is a sound approximation of P. The summary process is constructed from the following transformations to the program:

1) Consider the semantics of a call `spawn(f, T, L, C)` in the original program. This call launches a new process starting execution in function f with the initial set of tokens T, label L, and capabilities C. The function `spawn` is replaced with a function $\texttt{spawn}_\alpha$ that may non-deterministically launch a summary process or may launch a non-summary process at most once. The summary process p_{sum} maintains not only a label, but an under-over approximation pair of labels (U_l, O_l) that approximates all labels with which it has ever been initialized, with a similar pair (U_c, O_c) for capabilities. It also maintains an initial token set T_p that over-approximates all endpoints

4. Casewise, if $(\mathsf{label}(p) \subseteq_\alpha U) = \{1\}$, then the send is definitely successful. If $(\mathsf{label}(p) \subseteq_\alpha O) = \{0\}$, then the send is definitely not successful. Otherwise, the analysis assumes both may happen.

with which it has ever been initialized. The first call to spawn with label L and capabilities C initializes the approximations as $U_l = O_l = L$ and $U_c = O_c = C$. Each subsequent call to $\texttt{spawn}_\alpha$ then performs a *weak update* of these initialization fields. In other words, $\texttt{spawn}_\alpha(f, T, L, C)$ causes $U_l \leftarrow U_l \cap L$, $O_l \leftarrow O_l \cup L$ (similarly for U_c and O_c), and $T_p \leftarrow T_p \cup T$. The summary process begins execution on only the first call to $\texttt{spawn}_\alpha$ that chooses to launch the summary process.
2) When the summary process p_{sum} executes, it iterates an unbounded number of times over the code segment of p. At the beginning of each execution, it non-deterministically initializes its fields based on the current bounds held in the approximations of the initial state. In particular, it initializes its set of endpoint tokens to some T such that $T \subseteq T_p$, its label to some L such that $U_l \subseteq L \subseteq O_l$, and its capabilities to some C such that $U_c \subseteq C \subseteq O_c$.

For p_{sum} to be a sound approximation, it is assumed that exists some path of execution through the code of p_{sum}. Given that we replace every loop guard with a non-deterministic value, this can only not be the case when all paths contain a call to a function that is unconditionally infinitely recursive. In reality, this is not a serious concern as even if such a case does arise, the transformation can apply a simple code transformation that inserts a non-deterministic return at the beginning of the problematic functions.

4.5. Proof of Soundness

We now show that the above transformations applied to a Flume program result in a program is a sound approximation of the original program. Let P be a program that is the result of code abstraction, and let $\mathsf{Tr}(P)$ denote the set of all traces that can be generated by P. Let P have action alphabet A and let $\mathfrak{A}(P)$ have action alphabet A', where $\mathfrak{A}(P)$ is the result of applying the code and finite object abstractions to P. Let H be the set of all well-formed renaming functions from traces defined over A to traces defined over A'. It has already been shown that if some trace σ over A is a violation, then there will exist some $h \in H$ such that $h(\sigma)$ is a violation. To show that $\mathfrak{A}(P)$ is a sound approximation of P, it will thus be sufficient to show the following:

Lemma 4. *Let P, $\mathfrak{A}(P)$, and H be as defined above. For every trace $\sigma \in \mathsf{Tr}(P)$ and every renaming function $h \in H$ from the objects of P to those of $\mathfrak{A}(P)$, there exists some $\sigma' \in \mathsf{Tr}(\mathfrak{A}(P))$ that contains $h(\sigma)$ as a subsequence.*

Proof: Sketch. Use induction on the length of the trace $h(\sigma)$. For any execution in P that generates σ, there is a corresponding execution in $\mathfrak{A}(P)$ that executes the same sequence of Flume statements. Any mapping of channel values performed by some well-formed renaming function h is simulated by an allocation of channels in $\mathfrak{A}(P)$. The result of any label comparison, and thus the success of any action in the execution in P, is approximated by a corresponding action in the execution of $\mathfrak{A}(P)$. For the full proof, see the Appendix. □

Soundness of the abstraction then follows:

Theorem 1. *Let P be a program with abstraction $\mathfrak{A}(P)$. If $\mathfrak{A}(P) \models \varphi$, then $P \models \varphi$.*

Proof: We prove the contrapositive. Suppose that $P \not\models \varphi$. Then there exists some $\sigma \in \mathsf{Tr}(P)$ such that $\sigma \not\models \varphi$. Now, let A, A' be the action languages of P and $\mathfrak{A}(P)$ and let H be the set of renaming functions from A to A'. By Lem. 1, there exists some $h \in H$ such that $h(\sigma) \not\models \varphi$ and by Lem. 4, there exists some $\sigma' \in Tr(\mathfrak{A}(P))$ that contains $h(\sigma)$ as a subsequence. Thus by Lem. 2, $\sigma' \not\models \varphi$ and $\mathfrak{A}(P) \not\models \varphi$. Restated, if $\mathfrak{A}(P) \models \varphi$, then it must be the case that $P \models \varphi$. □

4.6. Implementing Sufficiently-Concrete

Recall from §3.5 that the heuristic for selecting the most "advantageous" concrete execution trace $\overline{\pi}$ was to make the selection so that in the abstraction of $\overline{\pi}$, the illegal information-flow began from a sufficiently-concrete process. We now define the instrumentation predicates that are added to the model so that the model checker will only consider such abstract traces.

Let $\mathsf{exclusively\_summary\_label}$ be a unary predicate over a label L defined as $L = \{t_s\}$. Let $\mathsf{exclusively\_summary\_cap}$ be defined similarly over capabilities C as $(C \neq \emptyset \land C \subseteq \{t_s^-, t_s^+\})$. Let $\mathsf{exclusively\_summary\_token\_set}$ be a unary predicate over a set of tokens T defined as $(\mathsf{endp}(T) \neq \emptyset \land \mathsf{endp}(T) \subseteq \{e_s^0, e_s^1\})$. Let $\mathsf{non\_summary}(p)$ be a unary predicate over a process that evaluates to true if p is a non-summary process. Let $\mathsf{sufficiently\_concrete}$ be defined as follows:

$$\begin{aligned}&\mathsf{sufficiently\_concrete}(p, (T, L, C)) = \\ &\quad \mathsf{non\_summary}(p) \\ &\quad \land \neg\mathsf{exclusively\_summary\_token\_set}(T) \\ &\quad \land \neg\mathsf{exclusively\_summary\_label}(L) \\ &\quad \land \neg\mathsf{exclusively\_summary\_cap}(C).\end{aligned}$$

The analysis instruments the program model to maintain the sufficiently-concrete mapping $\mathcal{M}_{\mathsf{sc}}$, which is a mapping from processes to their sufficient concreteness. $\mathcal{M}_{\mathsf{sc}}$ is updated at the spawn of a process p with spawn(f,T,L,C) as $\mathcal{M}_{\mathsf{sc}} := \mathcal{M}_{\mathsf{sc}}[p \mapsto \mathsf{sufficiently\_concrete}(p, (T, L, C))]$.[5]

5. When p is a summary process, $\mathsf{sufficiently\_concrete}(p, (T, L, C))$ is always false; hence, a strong update is sound.

5. Experiments

We implemented the described abstractions as a source-to-source translation using CIL [9], a front-end and analysis framework for C. The implementation takes as input a program written against the Flume API with type annotations that allow the code-abstraction translation to determine what variables are relevant to IPC flow. To analyze preexisting programs, we only needed to make a few minor modifications manually, including marking file descriptors as channels and translating calls to `fork` into equivalent calls to `spawn`. These changes were performed by hand for all programs discussed below except FlumeWiki.

The CIL transformation implements the code abstraction, finite-object abstraction, and sufficiently-concrete instrumentation on the input program. The information-flow properties are translated into an LTL formula over (abstract) executions of the model.

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.

FlumeWiki. FlumeWiki [3] is a Wiki based on the MoinMoin Wiki engine [20], but redesigned and implemented for the Flume API to ensure desired information-flow properties. A simplification of the design architecture for FlumeWiki serves as the basis for the running example in Fig. 1. FlumeWiki is unique among the test cases in that it is a rare application available for study with preexisting code written against the Flume API. We thus verified that the preexisting code for launching processes to service requests upholds the desired security and functionality properties. Our results, given in Tab. 1, demonstrate that we are able to verify properties for systems that operate over an unbounded number of processes, and do so in a reasonable amount of time.

FlumeWiki is implemented to ensure many information-flow properties. We focused on verifying the following:

- **Security:** Information from a Worker process that handles one request should never reach a process that handles another request.
- **Functionality:** A Worker process should always be able to send data to the Handler process.

We made the following noteworthy modifications to model the dispatch module of FlumeWiki:

- We modeled a requester that continuously loops, launching a Handler process to eventually launch a Worker to service a new request.
- We modeled each Worker as a compromised process that tries to create endpoints and send data to and receive data from all channels in the system. This models scenarios in which a malicious Worker attempts to create a channel and send data to it while another Worker attempts to read data from the same channel.

Apache. The Apache [10] webserver is structured so that a single module, the multi-process module, is responsible for launching processes for servicing requests. Apache can be configured to use as a multi-process module any of a handful of modules, each designed to take advantage of features of the host system and expected workload. We chose a module that implements a *preforking* scheme that preemptively launches a set of worker processes, each with its own channel for receving requests. We verified information-flow properties for the chosen multi-process module. While the properties are similar to those of FlumeWiki, our results indicate that our approach scales reasonably well, allowing us to analyze properties over unbounded processes for large-scale programs. We chose to verify the following properties for the module:

- **Security:** Information from a process that handles one request should never reach a process that handles another request.
- **Functionality:** A Worker process should always be able to send data to the source of the request.

Given that there is no preexisting Flume code for Apache, we wrote label manipulation code by hand and then verfied it automatically. Similar to FlumeWiki, we modeled Worker processes as compromised.

The analysis of Apache takes significantly longer than the analysis of programs whose model does not contain a summary process, and longer than FlumeWiki as well. We hypothesize that this is due to the fact that even the sliced form of Apache contains a more complicated control flow structure than FlumeWiki, complicating the verification problem for the model checker.

ClamAV. ClamAV [11] is a virus detection tool that periodically scans the file of a user, checking for the presence of viruses by checking the files against a database of virus signatures. We verified flow properties over the module that ClamAV uses to scan private files. Our results demonstrate that we are able to express and check a policy, *export protection*, that is significantly different from the policy checked for the server models. Furthermore, the results show that flow policies specified for systems executing over a bounded set of processes can be checked very quickly.

The checked properties are as follows:

- **Security:** ClamAV should never be able to send private information out over the network.
- **Functionality:** ClamAV should always be able to read data from private files.

To model this, we made the following manual changes to ClamAV:

- We constructed channels that represented a private file and the network. We instrumented `open` to non-deterministically open these files for access.
- We inserted points at which the ClamAV scanner non-deterministically becomes compromised.

Table 1. Results of abstraction and model checking.

Program	Property Verified?	Stmts.		Vars		Num procs	Processes in model		Model Extraction	Model Checking
		Orig.	Sliced	Orig.	Sliced	(runtime)	Non-sum.	Sum.		
FlumeWiki	√	110	34	61	39	unbounded	4	1	0.168s	36m 58s
Apache	√	596	149	327	39	unbounded	2	1	0.392s	1h 28m 45s
ClamAV	√	3427	1826	1377	695	2	2	0	1.135s	25s
OpenVPN	√	29494	8304	10933	3047	3	3	0	16.738s	59s

- The ClamAV scanner was wrapped similarly to as described in [2]. The wrapper creates and manipulates labels such that ClamAV is able to read the private files but not distribute them over an untrusted sink.

OpenVPN. OpenVPN [12] 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. The results of checking OpenVPN again demonstrate that the checker is able to verify properties over such programs very quickly. Moreover, they demonstrate that strong slicing allows us to perform whole-program analysis of real-world programs that rely on only a few key manipulations of relevant structures.

In particular, we checked OpenVPN against the following flow properties:

- **Security property:** 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.
- **Functionality property:** OpenVPN should always be able to access data from both networks.

We wrote a small manager that labeled channels representing each network and gave OpenVPN the ability to read data from each network. We then checked the flow properties against a system consisting of a manager process that launches OpenVPN.

6. Related Work

Our research builds on pre-existing work mainly from two topics: information-flow and software model checking. Much work has been done in developing interprocess information-flow systems, including the systems Asbestos [21], Hi-Star [2], and Flume [3]. While the mechanisms of these systems all 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.

There has been previous work in the static verification of information-flow systems. Multiple systems [18], [22] 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 with that enforced by interprocess DIFC systems, and they do not aim to reason about concurrent processes. The work that perhaps most closely resembles our own is that of EON [13] and [14]. EON 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 regarding the extraction of the model. The work in [14] analyzes the Flume system itself and formally proves a property of non-interference. In contrast, our approach focuses on automatically extracting and checking models of applications written for the Flume system and is based on predicate abstraction and model checking. It concerns verifying a different portion of the system stack and can be viewed as directly complementing that work.

Jaeger *et. al.* [23] present an approach to analyzing integrity protection in the SELinux example policy. Guttman *et. al.* [24] present a systematic way based on model checking to determine the information-flow security goals achieved by 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.* [25] describe an approach to the verication of LSM authorization hook placement using CQUAL, a type-based static-analysis tool.

We apply a model checker for concurrent C programs [26] that applies counter-example-guided predicate abstraction [17]. Furthermore, we improve the precision of concurrent model checking by soundly refining our model using the technique of random isolation [7]. There, random isolation was used to check atomic-set serializability problems. In this work, we use it to improve precision in reasoning about DIFC objects.

References

[1] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos operating

system," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 17–30, 2005.

[2] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 263–278.

[3] 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 '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 321–334.

[4] M. Conover, "Analysis of the Windows Vista Security Model," Symantec Corporation, Tech. Rep., 2008.

[5] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav, "Efficient verification of sequential and concurrent C programs," *Form. Methods Syst. Des.*, vol. 25, no. 2-3, pp. 129–166, 2004.

[6] A. Lal and T. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," in *Proc. Computer Aided Verification (CAV)*. Springer-Verlag, 2008.

[7] N. A. Kidd, T. W. Reps, J. Dolby, and M. Vaziri, "Finding concurrency-related bugs using random isolation," in *VMCAI*, 2009. [Online]. Available: \url{http://www.cs.wisc.edu/wpis/papers/vmcai09.pdf}

[8] E. M. Clarke and O. Grumberg, "Avoiding the state explosion problem in temporal logic model checking," in *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1987, pp. 294–303.

[9] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," 2002. [Online]. Available: citeseer.comp.nus.edu.sg/673175.html

[10] "Apache." [Online]. Available: http://www.apache.org

[11] "Clamav." [Online]. Available: http://www.clamav.net

[12] "Openvpn." [Online]. Available: http://www.openvpn.net

[13] A. Chaudhuri, P. Naldurg, S. K. Rajamani, G. Ramalingam, and L. Velaga, "EON: modeling and analyzing dynamic access control systems with logic programs," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 381–390.

[14] M. Krohn and E. Tromer, "Non-interference for a practical DIFC-based operating system," in *IEEE Symposium on Security and Privacy (to appear)*. IEEE Computer Society, 2009.

[15] C. Wright, C. Cowan, J. Morris, and S. S. G. Kroah-Hartman, "Linux security modules: general security support for the Linux kernel," in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, 2003, pp. 213–226.

[16] S. Graf and H. Saïdi, "Construction of Abstract State Graphs with PVS," in *CAV97*, ser. Lncs, O. Grumberg, Ed., vol. 1254. Haifa, Israel, June 22–25, 1997. New York, NY: Springer, June 1997, pp. 72–83.

[17] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[18] A. C. Myers, "JFlow: practical mostly-static information flow control," in *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1999, pp. 228–241.

[19] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar, "F-soft: Software verification platform," in *CAV '05*, 2005, pp. 301–306. [Online]. Available: http://dx.doi.org/10.1007/11513988_31

[20] MoinMoin, "The MoinMoin wiki engine," Dec. 2006. [Online]. Available: http://moinmoin.wikiwikiweb.de

[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.*, vol. 25, no. 4, p. 11, 2007.

[22] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[23] T. Jaeger, R. Sailer, and X. Zhang, "Analyzing integrity protection in the SELinux example policy," in *Proc. of the 11th USENIX Security Symposium*, 2003, pp. 59–74.

[24] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka, "Verifying Information-Flow Goals in Security-Enhanced Linux," *Journal of Computer Security*, 2005.

[25] X. Zhang, A. Edwards, and T. Jaeger, "Using CQUAL for static analysis of authorization hook placement," in *Proc. of the 11th USENIX Security Symposium*, 2002, pp. 33–48.

[26] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C," *Tse*, vol. 30, no. 6, pp. 388–402, June 2004.

Appendix

Trace preservation. *Let σ be a trace over an action alphabet $A = (C, P)$ where the process instances are partitioned into groups $G_0, G_1, \ldots, G_n$. Let $A' = (C', P')$ be a distinct action language with process instances partitioned into groups $G'_0, G'_1, \ldots, G'_n$ such that $\mathsf{entry}(G_i) = \mathsf{entry}(G'_i)$ and $|G'_i| = max(|G_i|, 2)$. If there exists some flow property φ such that $\sigma \not\models \varphi$, there exists some $h_s : S \to S'$ such that $h_s(\sigma) \not\models \varphi'$ where φ' is φ replaced with the corresponding process groups.*

Proof: We consider casewise the violations that σ may perform:

- Suppose that σ violates some security property $G \not\to^* H$. Because the channel-renaming function from which h_s is constructed is well-defined, the equality conditions over channels in subsequent actions in the trace are preserved. By our conditions for a well-formed process-renaming function, the groups of processes are preserved, so if $G \neq H$, then any well-formed renaming function h_s over traces will be such that $h_s(\sigma)$ violates the same security property. Now suppose that $G = H$, let p_i be the process instance that performs the initial send in the trace, and let p_f be the process that performs the final receive. In order for $\mathsf{transfer}(G, H, \sigma)$ to be true, it must be the case that $p_i \neq p_f$, so $|H| \geq |H'| = 2$. Thus there must exist some renaming h_d that maps the initial and final processes in σ to distinct processes in $h_d(\sigma)$.
- Suppose that σ violates some functionality property $G \to H$. Because every well-formed process renaming function preserves process groups, any such h_s will be such that $h_s(\sigma)$ violates the property.

$\square$

Soundness of code abstraction. *Let P be an original program and let $\mathfrak{C}(\cdot)$ perform the code abstraction defined in §4.3. Then for any flow property φ, if $\mathfrak{C}(P) \models \varphi$ then $P \models \varphi$.*

Proof: We will prove the contrapositive. Suppose that for some property φ, it is the case that $P \not\models \varphi$. Let an execution of P be represented as a sequence of statements. There must exist some $\sigma \in \mathsf{Tr}(P)$ generated by some execution E such that $\sigma \not\models \varphi$. In transforming P to $\mathfrak{C}(P)$, if we change a control-flow guard, then we weaken it to a nondeterministic value. Thus there is an execution E' through $\mathfrak{C}(P)$ that executes the same Flume statements executed by P. Furthermore, the manipulation of tags, labels, endpoints, and channels in $\mathfrak{C}(P)$ is unchanged from P. Thus the success of any send or recv is identical and E' generates a trace identical to σ, violating the flow property. $\square$

Lemma A.1. *Let L, M be labels over a set S of non-summary tags, let T be an independent set of tags that may contain at most one non-summary tag and one summary tag t_{sum}. Let $h_t : S \to T$ be one-to-one modulo $\{t_{\mathsf{sum}}\}$ and let h_l be the label-renaming function lifted from h_t. The result of the comparison $L \subseteq M$ is contained in $h_l(L) \subseteq_\alpha h_l(M)$.*

Proof: For $L \subseteq M$ and $h_l(L) \subseteq_\alpha h_l(M)$, consider the cases in the definition of $\subseteq_\alpha$. If either of the first two cases hold, then $L \subseteq_\alpha M = L \subseteq M$ and $(L \subseteq M) \in \{L \subseteq_\alpha M\}$. If the third case holds, then $h_l(L) \subseteq_\alpha h_l(M) = \{0, 1\}$ and it still must be the case that $(L \subseteq M) \in h_l(L) \subseteq_\alpha h_l(M)$. $\square$

Lemma 4. *Let P, $\mathfrak{A}(P)$, and H be as defined above. For every trace $\sigma \in \mathsf{Tr}(P)$ and every renaming function $h \in H$ from the objects of P to those of $\mathfrak{A}(P)$, there exists some $\sigma' \in \mathsf{Tr}(\mathfrak{A}(P))$ that contains $h(\sigma)$ as a subsequence.*

Proof: Define an *execution* E of P to be a serialized list of the statements obtained by interleaving the executions of all concurrent processes and let σ be the trace of E. Let the action alphabet of P be constructed as $A = (C, P)$, let the action alphabet of $\mathfrak{A}(P)$ be constructed as $A' = (C', P')$, and let $h_s : S \to S'$ be a well-formed renaming function from the trace language of the concrete program to that of the abstract program. Consider the renamed trace $h_s(\sigma)$. We will prove by induction on the length of $h_s(\sigma)$ that there exists a trace $\sigma' \in \mathsf{Tr}(\mathfrak{A}(P))$ such that $h_s(\sigma) \preceq \sigma'$. Although it is a stronger claim than what we need for the final result, we prove by induction that every action generated by a process p_i at a program point pp_j is generated by $h_p(p_i)$ at pp_j as well.

- $|\mathbf{h}(\sigma)| = \mathbf{0}$. In the trivial case, any execution of $\mathfrak{A}(P)$ generates a trace that contains $h(\sigma)$ as a subsequence.
- $|\mathbf{h}(\sigma)| = \mathbf{n} > \mathbf{0}$. In the inductive case, assume that there exists at least one execution that generates the prefix of $h(\sigma)$ of length $n - 1$. We now want to show that one of these executions generates the next action in $h(\sigma)$. Let p_i be the process instance in P, renamed to $h_p(p_i)$ in the renamed trace, that performs the next action in σ by reaching program point pp_0.

Part I. First, suppose that $h_p(p_i)$ is a non-summary process. We first show that there exists an execution in $\mathfrak{A}(P)$ that can reach pp_0 in its code segment. When E executed in P, it reached pp_0 and performed the last action in σ. Previously, it was either at a program point pp_1 where it performed its prior action or no such action existed and we define pp_1 to be the program point at the beginning of execution. In either case, the inductive hypothesis ensures that the process $h_p(p_i)$ reached pp_1. We have supposed that $h_p(p_i)$ is a non-summary process derived from p_i by code abstraction. It can thus execute any path through the code segment that p_i can and thus reach the program point pp_0 from the program point pp_1. Let $\mathcal{E}_0$ be the set of all executions where $h_p(p_i)$ does so.

Part II. We have thus established that for any execution in P, there is a non-empty set of executions in $\mathfrak{A}(P)$ that traverses in the same order all Flume statements retained from P. Each of these executions may vary in the values of its labels and endpoints, so we now need only prove that the Flume values of one of these executions cause its trace to match $h_s(\sigma)$. To see this, first note that the execution of E in P induces a one-to-one correspondence from invocations of `create_channel` to channel values, denoted as h_E. Furthermore, h_s is a well-formed renaming function over traces, and as such is constructed from a well-formed channel renamer h_c. It must be the case that h_c is one-to-one modulo $\{c_m\}$ for some c_m. Thus $h_c \circ h_E$ is a function from invocations of `create_channel` to C' that is one-to-one modulo $\{c_m\}$.
Now consider that an execution in $\mathcal{E}_0$ induces a mapping from invocations of `create_channel` to the set of channel values C' that is one-to-one modulo $\{c_{\mathsf{sum}}\}$ if c_{sum} is the summary channel. Furthermore, because the allocation of non-summary channels is nondeterministic, the set of executions in $\mathcal{E}_0$ induces all such mappings. Thus there exists a non-empty set of executions that induce the same mapping as $h_c \circ h_E$. Denote this set as $\mathcal{E}_1$.
We next consider whether, if the next action generated by the original program was a successful action or unsucessful, there exists an execution in the abstracted program with the same outcome. Let the call executed by p at program point pp_0 generate a generic action $\mathsf{action}(e)$ acting over an endpoint e. The success of this action depends completely on the comparison performed on $\mathsf{label}(e)$ and $\mathsf{label}(p)$. Note that each execution in P induces a one-to-one correspondence h_p from each concrete tag to the invocation of `create_tag` that creates it. Meanwhile, each execution in $\mathfrak{A}(P)$ induces a mapping h_a from invocations of `create_tag` to the tags that they return. This mapping is one-to-one modulo $\{t_{\mathsf{sum}}\}$ where t_{sum} is the summary tag. Thus the composition $h_a \circ h_p$ that maps the concrete tags used in P to the tags used in $\mathfrak{A}(P)$ is itself one-to-one modulo $\{t_{\mathsf{sum}}\}$. Thus by Lemma A.1, every label comparison $l \subseteq m$ occuring in the execution E is soundly approximated by a corresponding label comparison $l \subseteq_\alpha m$ for any execution in $\mathcal{E}_1$. The success of any action is determined completely by comparison over labels. Thus for any next action in $h_s(\sigma)$, there exists some execution in $\mathcal{E}_1$ that generates the same action.

Part III. We have established the inductive step in the case where the next action is generated by a process mapped to a non-summary process in $\mathfrak{A}(P)$. The case where the next action is to be performed by a summary process is similar. Again, we must determine that pp_0 is reachable by the summary process given that it has previously reached pp_1. However, pp_0 is certainly reachable from pp_1 given that the summary process may iterate over the program code an unbounded number of times. We now reason about the label and channel values at the program point. Suppose that the next action was executed originally by a process instance p_i with entry point f and given initial configuration c by `spawn`. In $\mathfrak{A}(P)$ there exist executions where the corresponding call to $\texttt{spawn}_\alpha(f, c)$ merges c with the approximation of all initial configurations. Thus according to the semantics described in §4.4.3, there will exist some iteration of the summary process that nondeterministically picks the same initial configuration c and executes under it. The success of every action in the execution of the process is completely determined by its initial configuration and the paths under which it executes. Given that there is an execution of $\mathfrak{A}(P)$ that starts with the same initial configuration as p_i and can thus execute any path that it may execute, we conclude that the action executed by p_i is executed by the summary process as well.

$\square$