Noninterference for a Practical DIFC-Based Operating System

Maxwell Krohn[1] and Eran Tromer[2]

[1] CyLab,
Carnegie Mellon University,
Pittsbrugh, PA, USA
krohn@mit.edu
[2] Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
Cambridge, MA 02139,
tromer@csail.mit.edu

Abstract. The *Flume* system is an implementation of decentralized information flow control (DIFC) at the operating system level. Prior work has shown Flume can be implemented as a practical extension to the Linux operating system, allowing real Web applications to achieve useful security guarantees. However, the question remains if the Flume system is actually secure. This paper compares Flume with other recent DIFC systems like Asbestos, arguing that the latter is inherently susceptible to certain wide-bandwidth covert channels, and proving their absence in Flume by means of a noninterference proof in the Communicating Sequential Processes formalism.

1 Introduction

Recent work in operating systems [1,2,3] makes the case that Distributed Information Flow Control (DIFC) [4] solves important application-level security problems for real systems. For example, modern dynamic web sites are trusted to safeguard private data for millions of users, but they often fail in their task (e.g., [5,6,7,8,9,10]). Web applications built atop DIFC operating systems can achieve better security properties, by factoring security-critical code into small, isolated, trustworthy processes, while allowing the rest of the application to balloon without affecting the TCB.

To achieve such a split between trustworthy and untrustworthy components, DIFC must monitor and regulate the information flows among them. Enforcement today exists in two forms: static, as an extension of a programming language's type system; and dynamic, as a feature of an OS's system call interface. These two styles have their strengths and weaknesses: DIFC for programming languages gives fine-grained guarantees about which parts of the program have been influenced by which types of data, but it requires rewrites of existing applications using one of a few compiled languages. DIFC at the OS level gives coarser-grained information flow tracking (each process is its own security domain) but supports existing applications and languages. In particular, popular Web applications written in popular interpreted languages (e.g. Python, PHP, Perl and Ruby) can achieve improved security on DIFC OSes. A case can be made for both techniques, but to date, only DIFC at the language level has enjoyed formal security guarantees.

This paper considers the Flume system: a DIFC OS implemented as a 30,000-line extension to a standard Linux kernel [3]. Flume allows legacy processes to run as before, while confining those that need strong security guarantees (like web servers and network applications) to a tightly-controlled sandbox, from which all of their communication must conform to DIFC-based rules. This technique produces real security improvements in real web applications, like the popular Python-based MoinMoin Wiki package. But all claims of application-level security presuppose a correct OS kernel, appealing to intuition alone to justify the OS's security. Intuition can mislead: other seemingly-secure OSes ([1,11]) have inadvertently included high-bandwidth covert channels in their very interface, allowing information to leak against intended security policies (see Section 3 for more details).

This paper presents the first formal noninterference security argument for a real DIFC operating system — in this case, Flume. A DIFC OS with provable security guarantees is an important foundation for provable application-level security, both in web services (Flume's first application) and in other security-sensitive applications.

The roadmap is follows. Section 2 reviews the Flume system and its intended policies at a high level: first and foremost, that untrustworthy applications can compute with private data without being able to reveal (i.e. leak) it. Section 3 describes potential pitfalls, motivating a formal approach. Section 4 describes the important parts of the Flume System using a formal process algebra, namely Communicating Sequential Processes (CSP). This model captures a trustworthy kernel, untrustworthy user-space applications, and user-space applications with privilege, which can selectively reveal or *declassify* sensitive data. Next, Section 5 proves that this model fulfills *noninterference*: that unprivileged user processes cannot leak data from the system, whether via explicit communication or implicit channels. Flume meets a CSP definition of noninterference that we have minimally extended to accommodate user-space declassifiers. Though the arguments focus on secrecy, the same model and proof also applies to integrity.

In sum, this paper contributes the following new results:

1. A formal model for a real DIFC Linux-based operating system, which captures a trustworthy kernel, and both privileged and unprivileged user-space applications; and
2. A formal proof of noninterference.

There are important limitations. First, the actual Flume implementation is not guaranteed to follow the model described in the process-algebra. Second, there are no guarantees that covert channels do not exist in parts of the system that the model abstracts. In particular, the Flume model does not capture physical hardware, so covert channels might of course exist in Flume's use of the processor, the disk, memory, etc. What our result does imply is that those operating systems the follow the given interface (like Flume) have a chance of achieving good security properties; i.e., wide leaks are not "baked" into their specifications. We leave a machine-checkable proof and a verified implementation of the model to future work.

2 Review of Flume

This section reviews Flume's security primitives, previously reported elsewhere [3]. Flume uses *tags* and *labels* to track data as it flows through a system. Let $\mathcal{T}$ be a very large set of opaque tokens called *tags*. A tag carries no inherent meaning, but processes generally associate each tag with some category of secrecy or integrity. For example, a tag $b \in \mathcal{T}$ may label Bob's private data.

Labels are subsets of $\mathcal{T}$. Labels form a lattice under the partial order of the subset relation [12]. Each Flume process p has two labels, S_p for secrecy and I_p for integrity. Both labels serve to (1) summarize which types of data have influenced p in the past and (2) regulate where p can read and write in the future. Consider a process p and a tag t. If $t \in S_p$, then the system conservatively assumes that p has seen some private data tagged with t. In the future, p can read more private data tagged with t but requires consent from an authority who controls t before it can reveal any data publicly. If there are multiple tags in S_p, then p requires independent consent for each tag before writing publicly. Process p's integrity label I_p serves as a lower bound on the purity of its influences: if $t \in I_p$, then every input to p has been endorsed as having integrity for t. To maintain this property, the system only allows p to read from other sources that also have t in their integrity labels. Files (and other objects) also have secrecy and integrity labels; they can be thought of as passive processes.

Distributed Information Flow Control (DIFC) is a generalization of centralized IFC. In centralized IFC, only a trusted "security officer" can create new tags, subtract tags from secrecy labels (*declassify* information), or add tags to integrity labels (*endorse* information). In Flume DIFC, any process can create new tags, which gives that process the privilege to declassify and/or endorse information for those tags.

2.1 Capabilities

Flume represents privilege using two *capabilities* per tag. Capabilities are objects from the set $\mathcal{O} = \mathcal{T} \times \{-, +\}$. For tag t, the capabilities are denoted t^+ and t^-. Each process p *owns* a set of capabilities $O_p \subseteq \mathcal{O}$. A process with $t^+ \in O_p$ *owns* the t^+ capability, giving it the privilege to add t to its labels; and a process with $t^- \in O_p$ can remove t from its labels. In terms of secrecy, t^+ lets a process add t to its secrecy label, granting itself the privilege to receive secret t data, while t^- lets it remove t from its secrecy label, effectively declassifying any secret t data it has seen. In terms of integrity, t^- lets a process remove t from its integrity label, allowing it to receive low-t-integrity data, while t^+ lets it add t to its integrity label, endorsing the process's current state as high-t-integrity. A process that owns both t^+ and t^- has *dual privilege* for t and can completely control how t appears in its labels. The set D_p where

$$D_p \triangleq \{t \mid t^+ \in O_p \ \wedge \ t^- \in O_p\}$$

represents all tags for which p has dual privilege.

Any process p can invent or "allocate" a new tag. Tag allocation yields a fresh tag $t \in \mathcal{T}$ and sets $O_p \leftarrow O_p \cup \{t^+, t^-\}$, granting p dual privilege for t. Tag allocation should not expose any information about system state.

For a set of capabilities $O \subseteq \mathcal{O}$, we define the notation:

$$O^+ \triangleq \{t \mid t^+ \in O\} \ , \quad O^- \triangleq \{t \mid t^- \in O\} \ .$$

2.2 Global Capabilities

Flume also supports a *global* capability set $\hat{O}$, Every process has access to every capability in $\hat{O}$, useful for implementing key security policies (see Section 2.4). A process p's effective set of capabilities is given by:

$$\bar{O_p} \triangleq O_p \cup \hat{O}$$

Similarly, its effective set of dual privileges is given by:

$$\bar{D_p} \triangleq \{t \mid t^+ \in \bar{O_p} \ \wedge \ t^- \in \bar{O_p}\}$$

Tag allocation can update $\hat{O}$; an allocation parameter determines whether the new tag's t^+, t^-, or neither is added to $\hat{O}$ (and thus to every current and future process's $\bar{O_p}$).

Flume restricts access to the shared set $\hat{O}$, lest processes manipulate it to leak data. A first restriction is that processes can only add a tag to $\hat{O}$ during the tag's allocation (otherwise a process p could leak information to a process q by either adding or refraining from adding a pre-specified tag to $\hat{O}$). A second restriction is that no process p can enumerate $\hat{O}$ or $\bar{O_p}$ (otherwise, p could poll $\|\hat{O}\|$ while q allocated new tags, allowing q to communicate bits to p). Processes can, however, enumerate their non-global capabilities (those in O_p), since they do not share this resource with other processes.

A process p can grant capabilities in O_p to process q so long as p can send a message to q. p can also subtract capabilities from O_p as it sees fit.

2.3 Security

The Flume model assumes many processes running on the same machine and communicating via messages, or "flows". The model's goal is to track data flow by regulating both process communication and process label changes.

Safe Label Changes In the Flume model (as in HiStar), the labels S_p and I_p of a process p can be changed only by an explicit request from p itself. Other models allow a process's label to change as the result of receiving a message [1,13,11], but implicit label changes turn the labels themselves into covert channels [12,2] (see Section 3). Only those label changes permitted by a process's capabilities are safe:

Definition 1 (Safe label change).
For a process p, let the label L be S_p or I_p, and let L' be the requested new value of the label. The change from L to L' is safe if and only if:

$$L' - L \subseteq (\bar{O}_p)^+ \quad \textit{and} \quad L - L' \subseteq (\bar{O}_p)^- \ .$$

For example, say process p wishes to subtract tag t from S_p, to achieve a new secrecy label S'_p. In set notation, $t \in S_p - S'_p$, and such a transition is safe only if p owns the subtraction capability for t (i.e. $t^- \in \bar{O}_p$). Likewise, t can be added only if $t^+ \in \bar{O}_p$.

Safe Messages Information flow control restricts process communication to prevent data leaks. The Flume model restricts communication among unprivileged processes as in traditional IFC: p can send a message to q only if $S_p \subseteq S_q$ ("no read up, no write down" [14]) and $I_q \subseteq I_p$ ("no read down, no write up" [15]).

Flume relaxes these rules to better accommodate declassifiers. Specifically, if two processes *could* communicate by changing their labels, sending a message using the centralized IFC rules, and then restoring their original labels, then the model can safely allow the processes to communicate without actually performing label changes. A process can make such a temporary label change only for tags in $\bar{D}_p$, i.e., those for which it has dual privilege. A process p with labels S_p, I_p would get maximum latitude in sending messages if it were to lower its secrecy to $S_p - \bar{D}_p$ and raise its integrity to $I_p \cup \bar{D}_p$. It could receive the most messages if it were to raise secrecy to $S_p \cup \bar{D}_p$ and lower integrity to $I_p - \bar{D}_p$.

The following definition captures these *hypothetical* label changes to determine what messages are safe:

Definition 2 (Safe message).
A message from p to q is safe iff

$$S_p - \bar{D}_p \subseteq S_q \cup \bar{D}_q \quad \textit{and} \quad I_q - \bar{D}_q \subseteq I_p \cup \bar{D}_p \ .$$

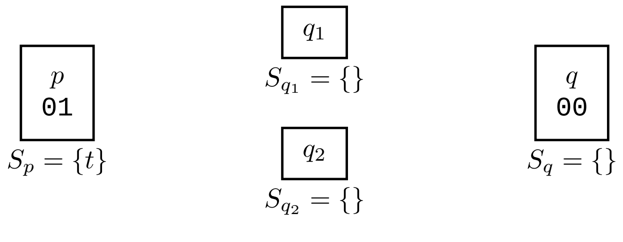

Fig. 1. The "leaking" system initializes.

External Sinks and Sources Any data sink or source outside of Flume's control, such as a remote host, the user's terminal, a printer, and so forth, is modeled as an unprivileged process x with permanently

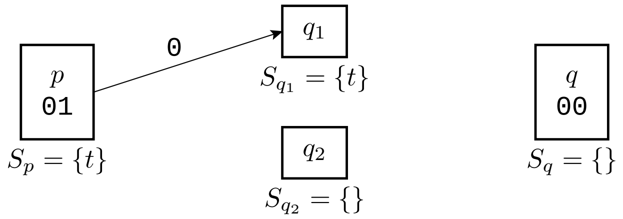

Fig. 2. p sends a "0" to q_i if the ith bit of the message is 0.

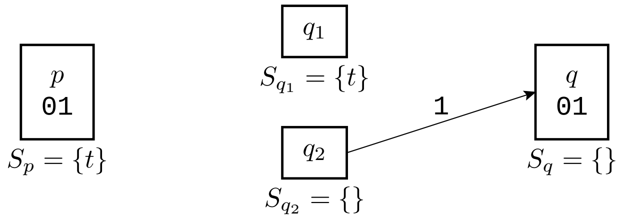

Fig. 3. If q_i did not receive a "0" before the timeout, it assumes an implicit "1" and writes "1" to q at position i.

empty secrecy and integrity labels: $S_x = I_x = \{\}$ and also $O_x = \{\}$. As a result, a process p can only write to the network or console if it could reduce its secrecy label to $\{\}$ (the only label with $S_p \subseteq S_x$), and a process can only read from the network or keyboard if it could reduce its integrity label to $\{\}$ (the only label with $I_p \subseteq I_x$).

2.4 Security Policies

The most important security policy in Flume is *export protection*, wherein untrustworthy processes can compute with secret data without the ability to reveal it. An export protection tag is a tag t such that $t^+ \in \hat{O}$ and $t^- \notin \hat{O}$. For a process p to achieve such a result, it creates a new tag t and grants t^+ to the global set $\hat{O}$, while closely guarding t^-. To protect a file f, p creates the file with secrecy label $\{t\}$. If a process q wishes to read f, it must first add t to S_q, which it can do since $t^+ \in \hat{O} \subseteq \bar{O}_q$. Now, q can only send messages to other processes with t in their labels. It requires p's authorization to remove t from its label or send data to the network. Additional Flume policies exist, like those that protect integrity.

3 Covert Channels in Dynamic Label Systems

As described in Section 2.3, processes in Flume change their labels explicitly; labels do not change implicitly upon message receipt, as they do in Asbestos [1] or IX [11]. We show by example why implicit label changes (also known as "floating" labels) enable high-throughput information leaks (as predicted by Denning [12]).

Consider a process p with secrecy label $S_p = \{t\}$ and a process q with $S_q = \{\}$, both with empty ownership sets $O_p = O_q = \{\}$. In a floating label system like Asbestos, p can send a message to q, and q will successfully receive it, upon which the kernel automatically raises $S_q = \{t\}$. Thus, the kernel can track which processes have seen secrets tagged with t, even if those processes are uncooperative. Such a scheme introduces new problems: what if a process q doesn't want its label to change from $S_q = \{\}$? For this reason, Asbestos also introduces "receive labels," which let processes filter out traffic and avoid unwanted label changes.

The problem with floating is best seen through example (see Figures 1–3). Imagine processes p and q as above, with a sender process p wanting to leak the 2-bit secret "`01`" to a receiver process q. Their goal is to transmit these bits without q's label changing. Figure 1 shows the initialization: q launches two helper processes (q_1 and q_2), each with a label initialized to $S_{q_i} = \{\}$. q's version of the secret starts out initialized to all `0`s, but it will overwrite some of those bits during the attack.

Next, p communicates selected bits of the secret to its helpers. If the ith bit of the message is equal to `0`, then p sends the message "`0`" to the process q_i. If the ith bit of the message is `1`, p does nothing. Figure 2 shows this step. When receiving this `0` bit, q_1's label changed, floating up from $\{\}$ to $\{t\}$, as the kernel accounts for how information flowed.

In the last step (Figure 3), the q_i processes wait for a predefined time limit before giving up. At the timeout, each q_i which did not receive a message (here, q_2) sends a message "`1`" to q, and upon receipt of this message q updates the bit at position i to 1. The remaining processes (q_1) do not write to q, nor could they without affecting q's label. Now, q has the exact secret, copied bit-for-bit from p. This example shows 2 bits of data leak, but by forking n processes, p and q can leak n bits per timeout period. Because Asbestos's *event process* abstraction makes forking very fast, this channel on Asbestos can leak kilobits of data per second.

This attack fails against the Flume system. In Figure 2, each q_i must each make a decision: should it raise its secrecy label to $S_{q_i} = \{t\}$, or leave it as is? If q_i raises S_{q_i} then it will receive messages from p, but it won't be able to write to q. Otherwise, q_i will never receive a message from p. In either case, q_i cannot alter its messages to q in response to messages from p. And crucially, q_i must decide whether to upgrade S_{q_i} *before* receiving messages from p.

4 The Formal Flume Model

That the above attack fails against the Flume model is useful intuition but proves nothing. This section and the next seek a formal separation between the Asbestos style of "floating" labels and the Flume style of "explicitly specified" labels. The ultimate goal is to prove that Flume exhibits *noninterference*: for example, that processes with empty ownership and whose secrecy label contains t cannot in any way alter the execution of those processes with empty labels. Such a noninterference result requires a formal model of Flume, which we build up here. Section 5 provides the proof that the Flume Model meets a standard definition of noninterference with high probability.

We present a formal model for the Flume System in the Communicating Sequential Processes (CSP) process algebra [16] (reviewed in Appendix .1). The model captures a kernel and arbitrary user processes that can interact through a system call interface. Processes can communicate with one another over IPC, changing labels, allocating tags, and forking new processes. The model dictates which kernel details are safe to expose to user-level applications, where I/O can safely happen, which return codes from system calls to provide, etc. It does not capture lower-level hardware details, like CPU, cache, memory, network or disk usage. Therefore, it is powerless to disprove the existence of covert channels that modulate CPU, cache, memory, network or disk usage to communicate data from one process to another.

Figure 4 depicts the Flume model organization. At a high level, the model splits each Unix-like process i running on a system (e.g., a web server or text editor) into two logical components: a *user half* U_i that can take almost any form, and a *kernel half* $i{:}K$ that behaves according to a strict state machine.[3] The user half of a process can communicate with its kernel half (and thus, indirectly, with other user processes) through the system call interface, which takes the form of a CSP channel $i.s$ between the U_i and $i{:}K$. The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (*TAGMGR*) manages the circulation of tags and globally-shared privileges;

[3] The CSP notation $i{:}K$ means the i-th instance of a template process K.

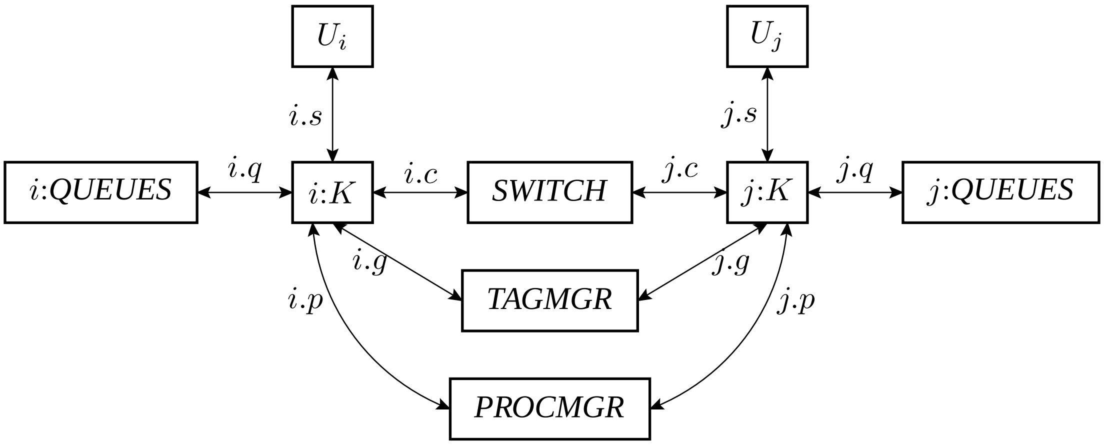

Fig. 4. Two user processes, U_i and U_j, in the CSP model for Flume. $i{:}K$ and $j{:}K$ are the kernel halves of these two processes (respectively), *TAGMGR* is the process that manages the global set of tags and associated privileges, *PROCMGR* manages the process ID space, and *SWITCH* enables all user-visible interprocess communication. Arrows denote CSP communication channels.

another global process (*PROCMGR*) manages the process ID space. The process *SWITCH* is involved with communication between user-level processes. The remainder of this section fills out the details of the Flume model.

4.1 System Call Interface

The "system-call" interface consists of events on the channel $i.s$ between U_i and $i{:}K$. Each user-level process has access to the following system calls:

- $t \leftarrow$ create_tag($which$)
  Allocate a new tag t, and depending on the parameter *which*, make the associated capabilities for t globally accessible. Here, *which* can be one of None, Remove or Add. For Remove, add t^- to $\hat{O}$, effectively granting it to all other processes; likewise, for Add, add t^+ to $\hat{O}$.
- rc $\leftarrow$ change_label($which, L$)
  Change the process's *which* label to L. Return Ok on success and Error on failure. Here, *which* can be either Secrecy or Integrity.
- $L \leftarrow$ get_label($which$)
  Read this process's own label out of the kernel's data structures. Here, *which* can be either Secrecy or Integrity, controlling which label is read.
- $O \leftarrow$ get_caps()
  Read this process's ownership set out of the kernel's data structures.
- send(j, msg, X)
  Send message *msg* and capabilities X to process j. (Crucially, the sender gets no indication whether the transmission failed due to label checks.)
- $(\mathrm{msg}, X) \leftarrow$ recv(j)
  Receive message *msg* and capabilities X from process j. Block until a message is ready.
- $j \leftarrow$ fork()
  Fork the current process; yield a process j. fork returns j in the parent process and 0 in the child process.

- $i \leftarrow$ getpid()
  Return i, the ID of the current process.
- drop_caps(X)
  Set $O_i \leftarrow O_i - X$.

See Appendix .5 for a description of the select system call that allows receiving processes to listen on multiple channels at once.

The Flume model places no restrictions on the U_is other than on their communication. A process U_i can communicate with $i{:}K$ via channel $i.s$; it can communicate with itself via channels that it invents; otherwise, it has no other communication channels.

4.2 Kernel Processes

Each process i has an instantiation of the kernel process K that obeys a strict state machine. We apply CSP's standard technique for "relabeling" the interior states of a process, giving $i{:}K$. By definition, $i{:}K$ and $j{:}K$ have different alphabets for $i \neq j$, so their operations cannot interfere. Each process $i{:}K$ takes on a state configuration based on process i's labels. That is, $i{:}K_{S,I,O}$ denotes the kernel half of process i, with secrecy label $S_i \subseteq \mathcal{T}$, integrity label $I_i \subseteq \mathcal{T}$, and ownership of capabilities given by $O_i \subseteq \mathcal{O}$.

At a high level, a kernel process K starts idle, then springs to life upon receiving an activation message. Once active, it receives either system calls from its user half, or internal messages from other kernel processes on the system. It eventually dies when the user process exits. In CSP notation:

$$K \;=\; b?(S, I, O) \rightarrow K_{S,I,O}$$

where b is the channel that K listens on for its "birth" message. It expects arguments of the form (S, I, O), to instruct it which labels and capabilities to start its execution with. Subsequently, $K_{S,I,O}$ handles the bulk of the kernel process's duties:

$$K_{S,I,O} \;=\; \mathit{SYSCALL}_{S,I,O} \mid \mathit{INTRECV}_{S,I,O}$$

where $\mathit{SYSCALL}$ is a subprocess tasked with handling all system calls, and $\mathit{INTRECV}$ is the *internal receiving* sub-process, tasked with receiving internal messages from other kernel processes.

Each system call gets its own dedicated subprocess:

$$\begin{aligned}\mathit{SYSCALL}_{S,I,O} \;=\; & \mathit{NEWTAG}_{S,I,O} \mid \\ & \mathit{CHANGELABEL}_{S,I,O} \mid \\ & \mathit{READMYLABEL}_{S,I,O} \mid \\ & \mathit{READMYCAPS}_{S,I,O} \mid \\ & \mathit{DROPCAPS}_{S,I,O} \mid \\ & \mathit{SEND}_{S,I,O} \mid \\ & \mathit{RECV}_{S,I,O} \mid \\ & \mathit{FORK}_{S,I,O} \mid \\ & \mathit{GETPID}_{S,I,O} \mid \\ & \mathit{EXIT}_{S,I,O}\end{aligned}$$

Section 4.4 presents all of these subprocesses in more detail.

4.3 Process Alphabets

In the next section, we will prove properties about the system, in particular, that messages between "high processes" (those that have a specified tag in their secrecy label) do not influence the activity of "low processes." The standard CSP approach to such proofs is to split the system's alphabet into two disjoint sets: "high" symbols, those that the secret influences; and "low" symbols, those that should not be affected by the secret. We must provide the appropriate alphabets for these processes so that any symbol in the model unambiguously belongs to one set or the other.

For example, take process i with secrecy label $S_i = \{t\}$ and integrity label $I_i = \{\}$. When U_i issues a system call (say $\mathsf{create\_tag(Add)}$) to its kernel half $i{:}K$, the trace for U_i is of the form

$$\langle \ldots, i.s!(\mathsf{create\_tag}, \mathsf{Add}), \ldots \rangle$$

and the trace for the kernel half $i{:}K$ is of the form

$$\langle \ldots, i.s?(\mathsf{create\_tag}, \mathsf{Add}), \ldots \rangle \ .$$

That is, U_i is sending a message $(\mathsf{create\_tag}, \mathsf{Add})$ on the channel $i.s$, and $i{:}K$ is receiving it. The problem, however, is that looking at these traces does not capture the fact that i's secrecy label contains t and therefore that U_i is in a "high" state in which it should not affect low processes. Such a shortcoming does not inhibit the accuracy of the model, but it does inhibit the proof of noninterference in Section 5.

A solution to the problem is to include a process's labels in the messages it sends. That is, once i has a secrecy label of $S = \{t\}$, its kernel process should be in a state such as $K_{\{t\},\{\},\{\}}$. When a kernel process is in this state, it will only receive system calls of the form $i.s?(\{t\}, \{\}, \{\}, \mathsf{create\_tag}, \mathsf{Add})$. Thus, U_i must now send system calls in the form:

$$i.s!(\{t\}, \{\}, \{\}, \mathsf{create\_tag}, \mathsf{Add})$$

In this regime, U_i must do its own accounting of S_i and I_i. If it fails to do so and sent a system call of the form

$$i.s!(\{\}, \{\}, \{\}, \mathsf{create\_tag}, \mathsf{Add})$$

the kernel would reject the call. Such a failure on U_i's part does not compromise security guarantees.

Messages of the form $c!(S, I, O, \ldots)$ and $c?(S, I, O, \ldots)$, for various channels c, occur often in our model. For concision, where S, I and O can be inferred from context in a kernel process $i{:}K_{S,I,O}$, we use this notation:

$$c \underset{K}{!} (x) \ \triangleq \ c!(S, I, O, x) \ , \qquad c \underset{K}{?} (x) \ \triangleq \ c?(S, I, O, x)$$

For example, kernel process $i{:}K_{\{t\},\{u\},\{t^-\}}$'s call to $s \underset{K}{?} (\mathsf{fork})$ is expanded as $s?(\{t\}, \{u\}, \{t^-\}, \mathsf{fork})$.

4.4 System Calls

We now define the sub-processes of $K_{S,I,O}$ that correspond to the kernel's implementation of each system call. The first system call subprocess handles a user process's request for new tags. Much of this system call is handled by a subroutine call to the global tag manager $TAGMGR$. After tag allocation, the kernel process transitions to a different state, reflecting the new privilege(s) it acquired for tag t.[4]

$$
\begin{aligned}
\mathit{NEWTAG}_{S,I,O} = \big(&s \underset{K}{?}(\mathsf{create\_tag}, w) \rightarrow \\
&g \underset{K}{!}(\mathsf{create\_tag}, w)?(t, O_{\text{new}}) \rightarrow \\
&s \underset{K}{!} t \;\rightarrow\; K_{S,I,O \cup O_{\text{new}}}\big)
\end{aligned}
$$

The *CHANGELABEL* subprocess is split into two cases, one for secrecy and one for integrity (the latter elided for brevity):

$$
\mathit{CHANGELABEL}_{S,I,O} = \; \mathit{S\text{-}CHANGE}_{S,I,O} \;\mid\; \mathit{I\text{-}CHANGE}_{S,I,O}
$$

$$
\begin{aligned}
\mathit{S\text{-}CHANGE}_{S,I,O} = \big(&\mathit{check} : \mathit{CHECK}_{S,I,O} /\!\!/ \\
&\big(s \underset{K}{?}(\mathsf{change\_label}, \mathsf{Secrecy}, S') \rightarrow \\
&\mathit{check}!(S, S') \rightarrow \mathit{check}?r \rightarrow \\
&\textbf{if } r \textbf{ then } s \underset{K}{!} \mathsf{Ok} \rightarrow K_{S',I,O} \\
&\qquad \textbf{else } s \underset{K}{!} \mathsf{Error} \rightarrow K_{S,I,O}\,\big)\big)
\end{aligned}
$$

In both cases, the user process specifies a new label, and the *CHECK* subroutine determines if that label change is valid. In the success case, the kernel process transitions to a new state, reflecting the new labels. In the failure case, the kernel process remains in the same state. The *CHECK* process computes the validity of the label change based on the process's current capabilities, as well as the global capabilities $\hat{O}$ (captured by a global process listening on the g channel):

$$
\begin{aligned}
\mathit{CHECK}_{S,I,O} =\, ?(L, L') \rightarrow& \\
&g!(\mathsf{check\text{-}}, L - L' - O^{-}) \;\rightarrow\; g?r \;\rightarrow \\
&g!(\mathsf{check+}, L' - L - O^{+}) \;\rightarrow\; g?a \;\rightarrow \\
&!(r \wedge a) \;\rightarrow\; \mathit{CHECK}_{S,I,O}
\end{aligned}
$$

As we will see below, the global tag register replies True to $(\mathsf{check\text{-}}, L)$ iff $L \subseteq \hat{O}^{-}$, and replies True to $(\mathsf{check+}, L)$ iff $L \subseteq \hat{O}^{+}$. Thus, we have that the user process can only change from label L to L' if it can subtract all tags in $L - L'$ and add all tags in $L' - L$, either by its own capabilities or those globally owned (see Definition 1 in Section 2.3).

The user half of a process can call the kernel half to determine its own S or I labels and its capabilities O:

$$
\begin{aligned}
&\mathit{READMYLABEL}_{S,I,O} = \\
&\quad \big(s \underset{K}{?}(\mathsf{get\_label}, \mathsf{Secrecy}) \rightarrow s \underset{K}{!} S \rightarrow K_{S,I,O} \mid \\
&\quad\; s \underset{K}{?}(\mathsf{get\_label}, \mathsf{Integrity}) \rightarrow s \underset{K}{!} I \;\rightarrow K_{S,I,O}\big)
\end{aligned}
$$

$$
\mathit{READMYCAPS}_{S,I,O} = \big(s \underset{K}{?}(\mathsf{get\_caps}) \rightarrow s \underset{K}{!} O \;\rightarrow K_{S,I,O}\big)
$$

[4] A note on notation: the channel between U_i and $i{:}K$ is named $i.s$ as stated earlier. However, the "$i.$" prefix is induced by the CSP renaming operator on the "template" kernel process K. In this section we define the subprocesses of K, so the channel is named merely s.

A process also can discard capabilities using *DROPCAPS*:

$$DROPCAPS_{S,I,O} = \big(s \underset{K}{?} (\mathsf{drop\_caps}, X) \;\to\; K_{S,I,O-X}\big)$$

On a successful drop of capabilities, the process transitions to a new kernel state, reflecting the reduced ownership set.

The next process to cover is forking. Recall that each active task i on the system has two components: a user component U_i and a kernel component $i{:}K$. The Flume model does not capture what happens to U_i when it calls fork,[5] but the kernel-side behavior of fork is specified as follows:

$$\begin{aligned} FORK_{S,I,O} = \big(s \underset{K}{?} (\mathsf{fork}) \;\to\; & \\ & p \underset{K}{!} (\mathsf{fork}, O) \to p?j \;\to \\ & s \underset{K}{!} j \;\to\; K_{S,I,O}\big) \end{aligned}$$

Recall that $i.p$ is a channel from the i-th kernel process to the process manager in the kernel, *PROCMGR*. (The latter allocates the child's process ID j and gives birth to $j{:}K$.)

The process handling getpid is straightforward:

$$\begin{aligned} GETPID_{S,I,O} = \big(s \underset{K}{?} (\mathsf{getpid}) \;\to\; & \\ & p!(\mathsf{getpid}) \to p?i \;\to \\ & s \underset{K}{!} i \;\to\; K_{S,I,O}\big) \end{aligned}$$

And user processes issue an exit system call as they terminate:

$$\begin{aligned} EXIT_{S,I,O} = \big(s \underset{K}{?} (\mathsf{exit}) \;\to\; & \\ & q!(\mathsf{clear}) \;\to\; p!(\mathsf{exit}) \;\to\; SKIP\big) \end{aligned}$$

Once a process with a given ID has run and exited, its ID is retired, never to be used again.

4.5 Communication

The communication subprocesses are the crux of the Flume CSP model. They require care to ensure that subtle state transitions in high processes do not result in observable behavior by low processes. At the same time, they must make a concerted effort to deliver messages, so that the system is useful.

The beginning of a message delivery sequence is the process $i{:}SEND_{S,I,O}$, invoked when U_i wishes to send a message to U_j. Messages are of the form (X, m), where X is a set of capabilities, and m is an arbitrary string. To send, the kernel shrinks the process's S label and grows its integrity label I as much as allowed by its privileges. The kernel also winnows the transmitted capabilities to those that U_i actually owns $(X \cap O)$. The message (X, m) passes through the switchboard process *SWITCH* via channel $i.c$, which forwards it j.

$$\begin{aligned} SEND_{S,I,O} = \big(s \underset{K}{?} (\mathsf{send}, j, X, m) \;\to\; & \\ & g!(\mathsf{dual\_privs}, O) \to g?D \to \\ & c!(S - D, I \cup D, j, X \cap O, m) \;\to\; K_{S,I,O}\big) \end{aligned}$$

[5] In Flume's concrete implementation of this model, forking has the familiar semantics of copying the address space and configuring the execution environment of the child process.

The process $SWITCH$ listens on the other side of the receive channel $i.c$. It accepts messages of the form $i.c?(S, I, j, X, m)$ and forwards them to the process $j{:}K$ as $j.c!(S, I, i, X, m)$:

$$\begin{aligned} SWITCH \;=\; & \,|_{\forall i}\big(i.c?(S, I, j, X, m) \;\rightarrow \\ & \qquad ((j.c!(S, I, i, X, m) \;\rightarrow SKIP) \;|||\; SWITCH)\big) \end{aligned}$$

The $SWITCH$ process sends messages in parallel with the next receive operation. This parallelism avoids deadlocking the system if the receiving process has exited, not yet started, or is waiting to send a message. In other words, the $SWITCH$ process is always willing to receive a new message, delegating potentially-blocking send operations to an asynchronous child process.

Once the message leaves the switch, the receiver process handles it with its $INTRECV$ subprocess. After performing the label checks given by Definition 2 in Section 2.3, this process enqueues the incoming message for later retrieval:

$$\begin{aligned} INTRECV_{S,I,O} \;=\; & c?(S_{\text{in}}, I_{\text{in}}, j, X, m) \;\rightarrow \\ & g!(\mathsf{dual\_privs}, O) \rightarrow g?D \rightarrow \\ & \textbf{if } (S_{\text{in}} \subseteq S \cup D) \;\wedge\; (I - D \subseteq I_{\text{in}}) \\ & \textbf{then } q!(\mathsf{enqueue}, (X, m)) \rightarrow K_{S,I,O} \\ & \textbf{else } K_{S,I,O} \end{aligned}$$

The final link in the chain is the actual message delivery in user space. For a user process to receive a message, it calls into the kernel, asking it to dequeue and deliver any waiting messages. Receiving also updates the process's ownership, to reflect new capabilities it gained.

$$\begin{aligned} RECV_{S,I,O} = \big(& s \underset{K}{?} (\mathsf{recv}, j) \;\rightarrow \\ & q!(\mathsf{dequeue}, j) \rightarrow q?(X, m) \;\rightarrow \\ & s \underset{K}{!} m \;\;\rightarrow\;\; K_{S,I,O \cup X}\big) \end{aligned}$$

4.6 Helper Processes

It now remains to fill in the details for the helper processes that the various $K_{S,I,O}$ processes call upon. They are: $TAGMGR$, which manages all global tag allocation and global capabilities; $QUEUES$, which manages receive message queues, one per process; and finally $PROCMGR$, which manages process creation, deletion, etc.

The Tag Manager (*TAGMGR*) The tag manager keeps track of a global universe of tags ($\mathcal{T}$), and the global set of privileges available to all processes ($\hat{O}$). It also tabulates which tags have already been allocated, so as never to reissue the same tag. The set $\hat{T}$ refers to those tags that were allocated in the past. Thus, the task manager's state is parameterized as $TAGMGR_{\hat{O},\hat{T}}$. Initially, $\hat{O}$ and $\hat{T}$ are empty:

$$TAGMGR = TAGMGR_{\{\},\{\}}$$

Once active, the tag manager services several calls:

$$
\begin{aligned}
\mathit{TAGMGR}_{\hat{O},\hat{T}} \;=\; & \mathit{NEWTAG+}_{\hat{O},\hat{T}} \mid \\
& \mathit{NEWTAG–}_{\hat{O},\hat{T}} \mid \\
& \mathit{NEWTAG0}_{\hat{O},\hat{T}} \mid \\
& \mathit{DUALPRIVS}_{\hat{O},\hat{T}} \mid \\
& \mathit{CHECK+}_{\hat{O},\hat{T}} \mid \\
& \mathit{CHECK\text{-}}_{\hat{O},\hat{T}}
\end{aligned}
$$

Many of these subprocesses will call upon a subroutine that randomly chooses an element from a given set. We define that subroutine here. Given a set Y:

$$\mathit{CHOOSE}_Y = ?(S, I, O) \;\rightarrow\; \textstyle\bigsqcap_{y \in Y} (!y) \;\rightarrow\; \mathit{STOP}$$

That is, the subprocess *CHOOSE* nondeterministically picks an element y from Y and returns it to the caller. As we will see in Section 5, *CHOOSE*'s refinement (i.e., its instantiation) has an important impact on security. It can, and in some cases should, take into account the labels on the kernel process on whose behalf it operates.

The first set of calls involve allocating new tags, such as:

$$
\begin{aligned}
\mathit{NEWTAG+}_{\hat{O},\hat{T}} = \mathit{choose}{:}\mathit{CHOOSE}_{\mathcal{T}-\hat{T}} /\!\!/ \\
\big|_{\forall i}\big(& i.g \underset{\mathcal{K}}{?} (\mathsf{create\_tag}, \mathsf{Add}) \;\rightarrow \\
& \mathit{choose}!(S, I, O)?t \;\rightarrow \\
& i.g!(t, \{t^-\}) \;\rightarrow \\
& \mathit{TAGMGR}_{\hat{O}\cup\{t^+\},\hat{T}\cup\{t\}}\big)
\end{aligned}
$$

That is, the subprocess *NEWTAG+* picks a channel i such that $i.g$ has input available. Then, it chooses an unallocated tag t via *CHOOSE* and returns that tag to the calling kernel process. It services the next request in a different state, reflecting that a new capability is available to all processes (t^+) and the tag t is now allocated.

We next define *NEWTAG–* and *NEWTAG0* similarly:

$$
\begin{aligned}
\mathit{NEWTAG–}_{\hat{O},\hat{T}} = \mathit{choose}:\mathit{CHOOSE}_{\mathcal{T}-\hat{T}} /\!\!/ \\
\big|_{\forall i}\big(& i.g \underset{\mathcal{K}}{?} (\mathsf{create\_tag}, \mathsf{Remove}) \;\rightarrow \\
& \mathit{choose}!(S, I, O)?t \;\rightarrow \\
& i.g!(t, \{t^+\}) \;\rightarrow \\
& \mathit{TAGMGR}_{\hat{O}\cup\{t^-\},\hat{T}\cup\{t\}}\big)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{NEWTAG0}_{\hat{O},\hat{T}} = \mathit{choose}:\mathit{CHOOSE}_{\mathcal{T}-\hat{T}} /\!\!/ \\
\big|_{\forall i}\big(& i.g \underset{\mathcal{K}}{?} (\mathsf{create\_tag}, \mathsf{None}) \;\rightarrow \\
& \mathit{choose}!(S, I, O)?t \;\rightarrow \\
& i.g!(t, \{t^-, t^+\}) \;\rightarrow \\
& \mathit{TAGMGR}_{\hat{O},\hat{T}\cup\{t\}}\big)
\end{aligned}
$$

The $DUALPRIVS$ subprocess gives a user process U_i access to the shared capabilities in $\hat{O}$. On input O_i, it returns $\bar{D}_i$, the set of tags for which process i has dual privilege. $DUALPRIVS$ is formulated as below to prevent user processes from enumerating the contents of $\hat{O}$. Since there are no tags t such that $\{t^-, t^+\} \subseteq \hat{O}$, the process must privately own at least one privilege for t to get dual privilege for it. Thus, $DUALPRIVS$ does not alert a process to the existence of any tags it did not already know of:

$$\begin{aligned} \mathit{DUALPRIVS}_{\hat{O},\hat{T}} = \big|_{\forall i}\big(& i.g?(\mathsf{dual\_privs}, O_i) \;\rightarrow \\ & i.g!((O_i^+ \cup \hat{O}^+) \cap (O_i^- \cup \hat{O}^-)) \;\rightarrow \\ & \mathit{TAGMGR}_{\hat{O},\hat{T}}\big) \end{aligned}$$

Finally, the behavior of $CHECK+$ has already been hinted at. Recall this subprocess checks to see if the supplied set of tags is globally addable:

$$\begin{aligned} \mathit{CHECK+}_{\hat{O},\hat{T}} = \big|_{\forall i}\big(& i.g?(\mathsf{check+}, L) \;\rightarrow \\ & (\textbf{if } L \subseteq \hat{O}^+ \\ & \;\textbf{then } i.g!\mathsf{True} \\ & \;\textbf{else } i.g!\mathsf{False}) \;\rightarrow \\ & \mathit{TAGMGR}_{\hat{O},\hat{T}}\big) \end{aligned}$$

And similarly:

$$\begin{aligned} \mathit{CHECK\text{-}}_{\hat{O},\hat{T}} = \big|_{\forall i}\big(& i.g?(\mathsf{check\text{-}}, L) \;\rightarrow \\ & (\textbf{if } L \subseteq \hat{O}^- \\ & \;\textbf{then } i.g!\mathsf{True} \\ & \;\textbf{else } i.g!\mathsf{False}) \;\rightarrow \\ & \mathit{TAGMGR}_{\hat{O},\hat{T}}\big) \end{aligned}$$

The Process Manager (*PROCMGR*) The main job of the process manager is to allocate process identifiers when kernel processes call fork. We assume a large space of process identifiers, $\mathcal{P}$. The process manager keeps track of subset $\hat{P} \subseteq \mathcal{P}$ to account for which of those processes identifiers have already been used. In then allocates from $\mathcal{P} - \hat{P}$.

$$\mathit{PROCMGR}_{\hat{P}} = \mathit{PM\text{-}FORK}_{\hat{P}} \mid \mathit{PM\text{-}GETPID}_{\hat{P}} \mid \mathit{PM\text{-}EXIT}_{\hat{P}}$$

To answer the fork operation, the process manager picks an unused process ID (j) for the child, gives birth to the child ($j{:}K$) with the message $j.b!(S, I, O)$, and returns child's process ID to the parent:

$$\begin{aligned} \mathit{PM\text{-}FORK}_{\hat{P}} = & \mathit{choose} : \mathit{CHOOSE}_{\mathcal{P}-\hat{P}} /\!\!/ \\ & \big|_{\forall i}\big(i.p?(S, I, O, \mathsf{fork}) \;\rightarrow \\ & \quad \mathit{choose}!(S, I, O)?j \;\rightarrow \\ & \quad j.b!(S, I, O) \;\rightarrow\; i.p!(j) \;\rightarrow \\ & \quad \mathit{PROCMGR}_{\hat{P} \cup \{j\}}\big) \end{aligned}$$

Trivially:

$$\mathit{PM\text{-}GETPID} = \big|_{\forall i}\big(i.p?(\mathsf{getpid})!i \;\rightarrow\; \mathit{PROCMGR}\big)$$

Kernel processes notify the process manager of their exits, which are handled as no-ops:

$$PM\text{-}EXIT = |_{\forall i}\big(i.p?(\mathsf{exit}) \;\rightarrow\; PROCMGR\big)$$

A final task for the process manager is to initialize the system, launching the first kernel process. This process runs with special process ID init, off-limits to other processes. Thus:

$$PROCMGR0 = \mathsf{init}.b!(\{\}, \mathcal{T}, \{\}) \rightarrow PROCMGR_{\mathcal{P}-\{\mathsf{init}\}}$$

See Appendix .2 for a description of the $QUEUES$ process, which is mostly an implementation detail.

4.7 High Level System Definition

The overall system SYS is an interleaving of all the processes specified. Recall that $\mathcal{P}$ is the set of all possible process IDs. The user-half of the system is:

$$UPROCS = \mathop{|||}_{j \in \mathcal{P}} U_j$$

The kernel processes are:[6]

$$KS = \mathop{|||}_{j \in \mathcal{P}} j{:}\Big(\big(K \mathop{\|}_{\{j.q\}} QUEUES\big) \setminus \alpha QUEUES\Big) \tag{1}$$

Adding in the helper process gives the complete kernel:

$$\begin{aligned}
KERNEL1 &= \big(KS \mathop{\|}_{\mathcal{P}.c} SWITCH\big) \setminus \alpha SWITCH \\
KERNEL2 &= \big(KERNEL1 \mathop{\|}_{\mathcal{P}.g} TAGMGR\big) \setminus \alpha TAGMGR \\
KERNEL &= \big(KERNEL2 \mathop{\|}_{\mathcal{P}.p} PROCMGR0\big) \setminus \alpha PROCMGR0
\end{aligned}$$

Finally:

$$SYS = UPROCS \mathop{\|}_{\mathcal{P}.s} KERNEL$$

This assembly of kernel process makes extensive use of the CSP hiding operator ("\"). That is, the combined process SYS does not show direct evidence of internal state transitions such as: communications between any $i{:}K$ and the switch; communications with the tag manager; communications with the process manager; etc. In fact the only events that remain visible are the workings of the user processes U_i and their system calls given by $i.s \underset{K}{?}$ and $i.s \underset{K}{!}$. By implication, kernels that implement the Flume model should hide the system's inner workings from unprivileged users (which is indeed the case for the Flume implementation). In practical terms, the CSP model for SYS shows what a non-root Unix user might see if examining his processes with the `strace` utility.

5 Noninterference

A mature definition in the literature for models like Flume's is *noninterference*. Informally [17]:

[6] Notation: As per standard CSP, αK denotes the "alphabet" of process K. Also, $\mathcal{P}.c \triangleq \{p.c \mid p \in \mathcal{P}\}$

> One group of users, using a certain set of commands, is *noninterfering* with another group of users if what the first group does with those commands has no effect on what the second group of users can see.

In terms of Flume, select any export-protection tag t, a process p with $t \in S_p$, and a process q with $t \notin S_q$. Noninterference means that p's execution path should be entirely independent of q's. If p could somehow influence q, then it could reveal to q information tagged with t, violating the export-protection policy.

5.1 Definition

We use Ryan and Schneider's definition of noninterference [18], where process equivalence follows the *stable failures model* [19,20]. This definition considers all possible pairs of traces for S that vary only by elements in the high alphabet (i.e., they are equal when projected to low). For each pair of traces, two experiments are considered: advancing S over the elements in left trace, and advancing S over the elements in the right trace. The two resulting processes must look equivalent from a "low" perspective. Formally:

Definition 3 (Noninterference for System S). *For a CSP process S, and an alphabet of low symbols $LO \subseteq \alpha S$, the predicate $\mathsf{NI}_{LO}(S)$ is true iff*

$$\forall tr, tr' \in traces(S) \;:\; tr \upharpoonright LO = tr' \upharpoonright LO \;\Rightarrow$$
$$\mathcal{SF}[\![S/tr]\!] \upharpoonright LO \;= \mathcal{SF}[\![S/tr']\!] \upharpoonright LO \;.$$

We say that the process S exhibits noninterference with respect to the low alphabet LO iff $\mathsf{NI}_{LO}(S)$ is true.

That is, after being advanced by tr and tr', the two processes must accept all of the same traces (projected to low) and refuse all of the same refusal sets (projected to low).

Given an arbitrary *export-protection* tag t (one such that $t^+ \in \hat{O}$ and $t^- \notin \hat{O}$), we define the high and low alphabets as follows. The high symbols emanate from a process with $t \in S_i$:

$$HI_t \triangleq \{i.b.(S, I, O, \dots) \mid t \in S\} \cup \{i.s.(S, I, O, \dots) \mid t \in S\}$$

The low symbols are the complement set:

$$LO_t \triangleq \{i.b.(S, I, O, \dots) \mid t \notin S\} \cup \{i.s.(S, I, O, \dots) \mid t \notin S\}$$

Stability and Divergence There are several complications. The first is the issue of whether or not the stable failures model is adequate. For instance, if a high process caused the kernel to diverge (i.e., *hang*), a low process could record such an occurrence on reboot, thereby leaking a bit (very slowly!) to low. By construction, the Flume kernel never diverges. One can check this property by examining each system call and verifying that only a finite number of internal events can occur before the process is ready to receive the next call. User-space process (e.g., U_i) can diverge, but they cannot observe each other's divergence, and so their divergence is inconsequential in our security analysis.

If divergence attacks were a practical concern, we could capture divergent behavior with the more general Failures, Divergences, Infinite Traces (FDI) model [20]. We conjecture that Flume's noninterference results under the stable failures model also hold in the FDI model, but the proof mechanics are yet more complicated.

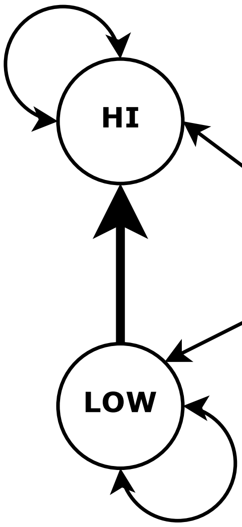

Fig. 5. Intransitive Noninterference. Arrows depict allowed influence. All influences are allowed *except* high to low.

Declassification The second complication is *declassification*, also known as *intransitive noninterference*. That is, the system should allow certain flows of information from "high" processes to "low" processes, if that flow traverses the appropriate declassifier. Figure 5 provides a pictorial representation: the system allows low processes and the declassifier to influence all other processes, and the high processes to influence other high processes and declassifiers but *not* to influence low processes. However, in the transitive closure, all processes can influence all other processes, which negates any desired security properties.

Previous work by Roscoe et al. assumes a global security policy, and modifies existing noninterference definitions to rule out flows not in that policy [21]. The most relevant definition is that of Bossi et al., which partitions the space of symbols into three sets — high, low, and declassify — and adjusts definitions in the CCS/SPA process algebra accordingly [22]. These extensions are not directly applicable in our setting, since Flume processes can dynamically transition between high, low and declassify states by creating new tags, receiving capabilities, changing labels, and dropping capabilities (all of the six transitions are possible).

To accommodate Flume's model we present a new definition, whose key distinction is to consider declassification orthogonally to secrecy. That is, at any given time, each process can be either high or low (determined by whether $t \in S$) and either *declassify* or *non-declassify* (determined by whether $t^- \in O$). We define a new set of symbols MID_t that emanate from or are received by declassifier processes. MID_t has the property that MID_t and HI_t need not be disjoint, nor do MID_t and LO_t:

$$\begin{aligned} MID_t \triangleq & \left\{ i.b.(S, I, O, \dots) \mid t^- \in O \right\} \cup \\ & \left\{ i.s.(S, I, O, \dots) \mid t^- \in O \right\} \end{aligned}$$

Next, consider all pairs of traces that differ only in high non-declassify elements, and are therefore equivalent when projected to $LO_t \cup MID_t$. Again, two experiments are performed, advancing S over the left and right traces. The two resulting processes must look equivalent from a low, non-declassify perspective. The final definition captures the notion that high, non-declassifying processes cannot interfere with low, non-declassifying processes. If a high process wishes to influence a low process, it must communicate through a declassifier process.

Definition 4 (Noninterference with declassification). *For a CSP process S, alphabets LO and MID contained in αS, the predicate $\mathsf{NID}_{LO}^{MID}(S)$ is true iff*

$$\forall tr, tr' \in traces(S) \;:\; tr \restriction (LO \cup MID) = tr' \restriction (LO \cup MID) \;\Rightarrow$$
$$\mathcal{SF}[\![S/tr]\!] \restriction (LO - MID) \;= \mathcal{SF}[\![S/tr']\!] \restriction (LO - MID) \;.$$

Note that this extended definition is equivalent to the standard definition if $MID = \{\}$, i.e., if declassification is disallowed. We conjecture an "unwound" version of this definition is equivalent to Bossi et al.'s DP_BNDC if HI_t and MID_t are disjoint, but we defer the proof to future work.

5.2 Allocation of Global Identifiers

The model presented in Section 4 is almost fully-specified, with an important exception: the process *CHOOSE*:

$$CHOOSE_Y = ?(S, I) \;\rightarrow\; \sqcap_{y \in Y} (!y) \;\rightarrow\; STOP$$

The "internal (nondeterministic) choice" operator ($\sqcap$) implies that the model requires further *refinement.* The question becomes: how to allocate tags and process identifiers?

An idea that does not work is sequential allocation, yielding the tag (or process ID) sequence $\langle 1, 2, 3, \ldots \rangle$. To attack this scheme, a low process forks, retrieving a child ID i. To communicate the value "k", the high process forks k times. The next time the low process forks, it gets process ID $i + k$, and by subtracting i recovers the high message. There are two problems: (1) low and high processes share the same process ID space; and (2) they can manipulate it in a predictable way.

The second weakness is exploitable even without the first. In a different attack, a high process communicates a "1" by allocating a tag via create_tag(Add), and communicates a "0" by refraining from allocating. If a low process could guess which tag was allocated (call it t), it could then attempt to change its label to $S = \{t\}$. If the change succeeds, then the low process had access to t^+ through $\hat{O}$, meaning the high process allocated the tag. If the change fails, the high process must not have allocated. The weakness here is that the low process "guessed" the tag t without the high process needing to communicate it. If such guesses were impossible (or very unlikely), the attack would fail.

Another idea—common to all DIFC kernels (c.f., Asbestos [1], HiStar [2] and the Flume implementation)—is random allocation from a large pool. The random allocation scheme addresses the second weakness—predictability—but not the first. That is, operations like process forking and tag creation always have globally observable side affects: a previously unallocated resource becomes claimed. Consider, as an example, this trace for the Flume system:

$$\begin{aligned} tr = \langle & i.b.(\{t\}, \{\}, \{\}, \{\}), \\ & i.s.(\{t\}, \{\}, \{\}, \mathsf{fork}), \\ & j.b.(\{t\}, \{\}, \{\}, \{\}), \\ & i.s.(\{t\}, \{\}, \{\}, j), \ldots \rangle \end{aligned}$$

A new process i is born, with secrecy label $S_i = \{t\}$, and empty integrity and ownership. Thus, i's actions fall into the HI_t alphabet. Once i starts, it forks a new process, which the kernel randomly picks as j. The child j runs with secrecy $S_j = \{t\}$, inheriting its parent's secrecy label.

Projecting this trace onto the low alphabet yields the empty sequence ($tr \restriction LO_t = \langle\rangle$). Thus, this trace should have no impact on the system from a low process k's perspective. Unfortunately, this is not the case. Before tr occurred, l could have forked off process j, meaning:

$$\begin{aligned} tr' = \langle & k.b.(\{\},\{\},\{\},\{\}), \\ & k.s.(\{\},\{\},\{\},\mathsf{fork}), \\ & j.b.(\{\},\{\},\{\},\{\}), \\ & k.s.(\{\},\{\},\{\},j), \ldots\rangle \end{aligned}$$

was also a valid trace for the system. But after tr occurs, tr' is no longer possible, since the process j can only be born once. In other words, $tr \frown tr'$ is not a valid trace for the system but tr' is by itself. This contradicts the definition of noninterference in the stable failures model of process equivalence.

To summarize, we have argued that allocation of elements from $\hat{O}$ and $\hat{P}$, must obey two properties: (1) unpredictability and (2) partitioning. Our approach is to design a randomized allocation scheme that achieves both. Define parameters:

$$\begin{aligned} \alpha &\triangleq \log_2(\text{the number of tags}) \\ \beta &\triangleq \log_2(\text{maximum number of operations}) \\ \epsilon &\triangleq -\log_2(\text{acceptable failure probability}) \end{aligned}$$

A reasonable value for β is 80, meaning that no instance of the Flume system will attempt more that 2^{80} operations. Since tag allocation, forking and constructing labels count as operations, the system expresses fewer than 2^{β} tags, process IDs, or labels in its lifetime. A reasonable value for ϵ is 100, meaning the system fails catastrophically at any moment with probability at most 2^{-100}.

Define a lookup table $s(\cdot)$, that given any label or capability set outputs a integer in $[0, 2^{\beta})$ that uniquely identifies it. This serialization can be predictable. Next consider the family of all *injective* functions:

$$G : (\{0,1\}^{\beta}, \{0,1\}^{\beta}, \{0,1\}^{\beta}, \{0,1\}^{\beta}) \to \{0,1\}^{\alpha}$$

The Flume system, upon startup, picks an element $g \in G$ at random. When called upon by a process with labels S, I, O to allocate a new tag or process ID, it returns $g(s(S), s(I), s(O), x)$, for some heretofore unused $x \in \{0,1\}^{\beta}$. The output is a tag in $\{0,1\}^{\alpha}$. Appendix .3 derives $\alpha \geq \max(\epsilon + 1, 4\beta)$, meaning $\alpha = 320$ for our example parameters.

Thus, we let $\mathcal{T} = \mathcal{P} = \{0,1\}^{\alpha}$, for a sufficiently large α. The kernel picks $g \in G$ at random upon startup. Then *CHOOSE* is refined as:

$$\mathit{CHOOSE}_Y = ?(S, I, O) \;\rightarrow\; \textstyle\bigsqcap_{y \in \mathcal{G}(S,I,O,Y)} (!y) \;\rightarrow\; \mathit{STOP}$$

where

$$\mathcal{G}(S, I, O, Y) = \left\{ t \mid x \in \mathcal{T} \wedge t = g(s(S), s(I), s(O), x) \wedge t \in Y \right\}.$$

Note that $\mathcal{G}(S, I, O, Y) \subseteq Y$, so the nature of the refinement is just to restrict the set of IDs that CHOOSE_Y will ever output, based on the capabilities, secrecy and integrity labels of the calling process.

5.3 Theorem and Proof

The main theorem is as follows:

Theorem 1 (Noninterference in Flume). *For any security parameter ϵ, there exists an instantiation of CHOOSE such that: for any export-protection tag t, for any Flume instance SYS,* $\Pr[\mathsf{NID}^{\mathit{MID}_t}_{\mathit{LO}_t}(\mathit{SYS})] \geq 1 - 2^{-\epsilon}$.

In other words, the system administrator of a Flume system first decides on a security parameter ϵ, expecting calamitous system collapse with probability of at most $1 - 2^{-\epsilon}$. He instantiates *CHOOSE* with ϵ, then boots the system. The init process runs, spawning an assortment of user processes, which combine with the kernel processes to constitute a new overall system *SYS*. For any export protection tag t that is allocated *as the system runs*, the extended noninterference property holds with the desired probability. This guarantee holds over all instances of *SYS*, regardless of what user processes (i.e., *UPROCS*) malicious users might cook up.

The proof is by induction over the number of low symbols in the two traces, tr and tr'. (Recall that tr and tr' are equivalent when projected to the low/mid alphabets). For the base case, tr and tr' have no low symbols, and therefore have no high symbols, since the system accepts only low symbols in its initial state. Since $tr = tr' = \{\}$, the theorem follows trivially.

We prove the inductive step casewise, considering each system call and whether the kernel is looking to accept a new system call or reply to an outstanding call. Most cases reason about the causal relationships among events in the trace. A more involved case is change_label, which must consider the unlikely case that a low process guessed which tags a high process received from the kernel when calling create_tag. See Appendix .4 for details.

6 Discussion

To review, we have described the Flume kernel both informally and with CSP formalism and proven that the CSP model upholds a definition of noninterference. In this section, we discuss the implications of these results, and how they can be translated to a practical system.

6.1 Refinement

Due to the well-known *refinement paradox*, the Flume model might satisfy noninterference, but implementations (i.e. *refinements*[7]) of it might not [23]. To circumvent this paradox, Lowe has recently strengthened notions of noninterference, requiring that a system like *SYS* and all of its refinements exhibit noninterference [24]. Other work has suggested restricting refinement to a set of operators known to preserve security guarantees [25]. We follow Lowe's approach as best as possible, arguing that noninterference holds for most refinements.

The parts of the Flume model that need refinement are those that display non-determinism via the $\sqcap$ or $\Box$ operators: (1) the *CHOOSE* process; (2) the user processes U_i; (3) "scheduling"; and (4) the *tock* events in timed CSP. As for (1), the proof in Section .4 holds for some refinements of *CHOOSE*, such as the random function specified in Section 5.2 and a more practical hash-based approach considered below. A Flume implementation should refine *CHOOSE* as specified, or with a method known to preserve noninterference. As for (2), the proof in Section .4 holds for *arbitrary* refinements of user processes U_is, as long as they communicate only through the designated system calls (see Section 4.1). In practice, we cannot hope to isolate the U_is completely from one another: they can communicate by manipulating shared hardware resources (e.g., disks, CPU cache, CPU cycles, network bandwidth, etc.)

As for (3), the Flume model hides scheduling for simplicity: any interleaving of processes is admissible, by Equation 1 in Section 4.7. However, a practical refinement of Flume would implement "fairness" restrictions on scheduling, disallowing one process from consuming more than its "fair" share of resources. Scheduling refinements, in and of themselves, do not affect the proof of security: noninterference holds in any reordering of high and low processes, as long as all processes get to run eventually, and do not have any comprehension of time. As for (4), Appendix .5 explores extending the Flume model to show an explicit

[7] Q is a refinement of P iff $\mathcal{SF}[\![Q]\!] \subseteq \mathcal{SF}[\![P]\!]$.

passage of time: the event *tock* denotes one clock tick, and all parts of the model must stand aside and yield to *tock*. We conjecture that if $tock \in HI_t$, then the proof holds for all refinements of the scheduler, and that if $tock \in LO_t$, that the proof holds for only some refinements. Further exploration is deferred to future work.

In sum, we believe the Flume model to maintain noninterference under important refinements—tag allocation, arbitrary user processes, and scheduling—if timing channels are excluded (i.e., if $tock \in HI_t$). Of course, this is a far cry from proving noninterference in a working implementation, but a significant improvement over the status quo. Systems such as Asbestos and IX have gaping covert channels baked into their very specifications, so *any* refinements of those systems are insecure. By contrast, an OS developer has a fighting chance to realize a secure Flume implementation.

6.2 Kernel Organization

The Flume DIFC model is a "monolithic" kernel design, in which the kernel is a hidden black box, and user-level processes have a well-specified system call interface. Some modern approaches to kernel design (e.g. the Exokernel [26] and the Infokernel [27]) expose more of the kernel's inner workings to give application developers more flexibility. However, such transparency in an information-flow control setting can leak information: imagine a high process issuing create_tag, and a low process observing *TAGMGR*'s transitions. The simplest way to work around this problem is to conceal the inner workings of the kernel (as we have done). Another, more complicated solution, is to model more parallelism inside the kernel, so that the tag manager can serve both high and low concurrently.

The Flume model captures most of the kernel processes—like the $i{:}K$, the tag manager, and the process manager—as single-threaded processes. For instance, if the tag manager is responding to a request for $i.g.(\mathsf{create\_tag}, w)$, it cannot service $j.g.(\mathsf{create\_tag}, w)$ until it replies to $i.g.(\mathsf{create\_tag}, w)$. In practical implementations of this CSP model, such serialization might be a bottleneck for performance. More parallelism internal to the kernel is possible, but would require explicit synchronization through locks, and more complexity overall.

6.3 Tag Allocations

The use of a truly random function for *CHOOSE* is impractical, as are tag sizes of 320 bits. In practice, a weaker cryptographic primitive suffices, such as a Message Authentication Code (MAC) [28] with a collision-resistant hash function [29]. Let M be such a MAC of suitable input length. The kernel picks a random secret key k on startup, and then computes new tags and process IDs using $M_k(S, I, O, x)$ for a counter variable x. This construction approximates both important properties. The unforgability property of the MAC implies that an adversary cannot find $(S, I, O, x) \neq (S', I', O', x')$ such that $\mathrm{HMAC}_k(S, I, O, x) = \mathrm{HMAC}_k(S', I', O', x')$, so a high process with secrecy $\{t\}$ and a low process with secrecy $\{\}$ will not get the same tag. Similarly, user processes cannot predict the output of $\mathrm{HMAC}_k(S, I, O, x)$ without knowing k.

6.4 Integrity

Though we have focused on secrecy, the same arguments hold for integrity. Analogously, one would pick an *integrity-protection* tag t, one for which $t^- \in \hat{O}$ and $t^+ \notin \hat{O}$. The low symbols are those whose integrity tags contain t, and the high symbols are those that do not. The same proof shows that the high events do not interfere with the low.

7 Related Work

Information flow control (IFC) at the operating system level dates back to the centralized military systems of the 70s, and 80s [30,15,31]. Several systems like IX [11] and SELinux [32] integrated information-flow ideas with Unix-like operating systems in 90s. Denning first pointed out that dynamically-adjusted security labels could leak data [12] and suggested instead static checking, which later found fruition as type-analysis [33]. *Decentralized* declassification and endorsement proved a key relaxation, making IFC practical for language-based systems [4], and eventual spurring a revitalization of the idea in operating systems and web-serving settings with the Asbestos [1], HiStar [2] and Flume [3] systems. HiStar introduced the idea of "self-tainting," solving the wide covert channel described in Section 3. Flume later adopted a similar strategy, but within a streamlined label system.

Taint-tracking is another technique for tracking information flow through legacy software written in arbitrary languages [34,35]. Such systems run a target application as rewritten binary, without the cooperation or recognition of the application in question, meaning they must infer label changes. Therefore taint-tracking systems are susceptible to the covert channel attacks described in Section 3, and cannot uphold noninterference.

Goguen and Meseguer introduced the idea of noninterference for security protocols [17], while Volpano et al. first showed that type systems could provably uphold the idea [36]. More recently, Zheng and Myers [37] and also Tse and Zdancewic [38] proved that statically-typed systems with runtime principles could still obey noninterference. Relative to Flume, information flow is monitored at a finer granularity. On the other hand, the Flume model provides more flexibility as to how the various user processes U_i behave: it restricts these processes from accessing all but one communication channel, but otherwise they can act arbitrarily and need not be type-checked.

The proofs offered here are manual. In future work, we hope to investigate shaping the Flume model used in the proof to a form that is amenable to automated analysis. Lowe first used an automated checker to break protocol previous assumed secure [39]. Ryan and Schneider also describe how the FDR automated checker can verify standard security protocols [40],

8 Conclusion

This paper presented the first formal security argument for a DIFC-based operating system. It modelled Flume using the CSP formalism, and proved that the model fulfills noninterference—an end-to-end property that protects secrecy and integrity even against subtle covert channel attacks. The model and proof are not substantially weakened by the refinement paradox, since the proof holds for many refinements of the model. Future work calls for further investigation of timing-based covert and side channels, and automation of the proof techniques, for both the model and its implementation.

Acknowledgments

Maxwell Krohn was supported at MIT CSAIL by the joint NSF CyberTrust/DARPA grant CNS-0430425, Nokia, and an NSF Graduate Student Fellowship. This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. Eran Tromer was supported by NSF CyberTrust grant CNS-0808907 and AFRL grant FA8750-08-1-0088. Views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, CMU, AFRL, or the U.S. Government or any of its agencies. Much of the material in this paper appeared in Maxwell Krohn's doctoral thesis, advised by Frans Kaashoek, Robert Morris and Eddie Kohler, and read by Butler Lampson. We thank them all for their advise, careful reading and comments. Thanks to Greg Morrisett, Andrew Myers, Nickolai Zeldovich, Alex Yip, Micah Brodsky, Adrian Perrig, and Anupam Datta for their ideas and guidance. We thank the anonymous reviewers for their detailed comments and suggestions.

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," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.
2. N. B. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.
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 *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.
4. A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malô, France, Oct. 1997, pp. 129–142.
5. J. Fielding, "UN website is defaced via SQL injection," *Tech Republic*, Aug. 2007, `http://blogs.techrepublic.com.com/networking/?p=312`.
6. R. Lemos, "Payroll site closes on security worries," *Cnet News.com*, Feb. 2005, `http://news.com.com/2102-1029_3-5587859.html`.
7. News10, "Hacker accesses thousands of personal data files at CSU Chico," Mar. 2005, `http://www.news10.net/display_story.aspx?storyid=9784`.
8. K. Poulsen, "Car shoppers' credit details exposed in bulk," *SecurityFocus*, Sept. 2003, `http://www.securityfocus.com/news/7067`.
9. ——, "FTC investigates petco.com security hole," *SecurityFocus*, Dec. 2003, `http://www.securityfocus.com/news/7581`.
10. R. Trounson, "Major breach of UCLA's computer files," *Los Angeles Times*, Dec. 12 2006.
11. M. D. McIlroy and J. A. Reeds, "Multilevel security in the UNIX tradition," *Software—Practice and Experience*, vol. 22, no. 8, pp. 673–694, 1992.
12. D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.
13. T. Fraser, "LOMAC: Low water-mark integrity protection for COTS environments," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000, pp. 230–245.
14. D. E. Bell and L. J. LaPadula, "Secure computer system: Unified exposition and Multics interpretation," MITRE Corparation, Bedford, MA, Tech. Rep. MTR-2997, Rev. 1, March 1976.
15. K. J. Biba, "Integrity considerations for secure computer systems," MITRE Corp., Bedford, MA, Tech. Rep. MTR-3153, Rev. 1, 1976.
16. C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, New Jersey: Prentice/Hall International, 1985.
17. J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1982.
18. P. A. Ryan and S. A. Schneider, "Process algebra and non-interference," *Journal of Computer Security*, no. 9, pp. 75–103, 2001.
19. A. W. Roscoe, *A Theory and Practice of Concurrency*. London, UK: Prentice Hall, 1998.
20. S. Schneider, *Concurrent and Real-Time Systems: The CSP Approach*. Chichester, UK: John Wiley & Sons, LTD, 2000.
21. A. Roscoe and M. H. Goldsmith, "What is intransitive noninterference?" in *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 1999.
22. A. Bossi, C. Piazza, and S. Rossi, "Modelling downgrading in information flow security," in *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, June 2004.

23. J. Jacob, “On the derivation of secure components,” in *Proceedings of the the IEEE Symposium on Security and Privacy*, Oakland, CA, 1989.
24. G. Lowe, “On information flow and refinement-closure,” in *Proceedings of the Workshop on Issues in the Theory of Security (WITS07)*, 2007.
25. H. Mantel, “Preserving Information Flow Properties under Refinement,” in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.
26. D. R. Engler, M. F. Kaashoek, and J. O’Toole, “Exokernel: An operating system architecture for application-level resource management,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, Dec. 1995, pp. 251–266.
27. A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici, “Transforming policies into mechanisms with infokernel,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, Lake George, New York, Oct. 2003, pp. 90–105.
28. M. Bellare, R. Canetti, and H. Krawczyk, “Keying hash functions for message authentication,” in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, Aug. 1996, pp. 1–15.
29. FIPS 180-2, *Secure Hash Standard*, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, Aug. 2002.
30. D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” MITRE Corparation, Bedford, MA, Tech. Rep. Technical Report 2547, Volume I, Mar. 1973.
31. *Trusted Computer System Evaluation Criteria (Orange Book)*, DoD 5200.28-STD ed., Department of Defense, December 1985.
32. P. Loscocco and S. Smalley, “Integrating flexible support for security policies into the Linux operating system,” in *Proceedings of the 2001 USENIX Annual Technical Conference*, San Diego, CA, June 2001, FREENIX track.
33. J. Palsberg and P. Ørbæk, “Trust in the λ-Calculus,” in *Proceedings of the 2nd International Symposium on Static Analysis*, Sept. 1995.
34. J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” in *Network and Distributed System Security Symposium (NDSS)*, Feb. 2005.
35. S. McCamant and M. Ernst, “Quantitative Information Flow as Network Flow Capacity,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.
36. D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” vol. 4, no. 2-3, 1996, pp. 167–187.
37. L. Zheng and A. C. Myers, “Dynamic security labels and noninterference,” in *Proceedings of the 2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, Aug. 2004.
38. S. Tse and S. Zdancewic, “Run-time principals in information-flow type systems,” *ACM Transactions on Programming Language Systems*, vol. 30, no. 1, 2007.
39. G. Lowe, “Some new attacks upon security protocols,” in *Proceedings of the 9th IEEE Computer Security Foundations Workshop (CSFW)*, 1996.
40. P. Ryan and S. Schneider, *The Modelling and Analysis of Security Protocols: The CSP Approach*. Addison-Wesley Professional, 2001.
41. J. Ouaknine, “A framework for model-checking timed CSP,” Oxford University, Tech. Rep., 1999.

.1 Review of CSP

Communicating Sequential Processes (CSP) is a process algebra useful in specifying systems as a set of parallel state machines that sometimes synchronize on events. We offer a brief review of it here, borrowing heavily from Hoare’s book [16].

CSP Processes Among the most basic CSP examples is Hoare’s vending machine:

$$VMS = in25 \rightarrow choc \rightarrow VMS$$

This vending machine waits for the event $in25$, which corresponds to the input of a quarter into the machine. Next, it accepts the event $choc$, which corresponds to a chocolate falling out of the machine. Then it returns to the original state, with a recursive call to itself. The basic operator at use here is the *prefix* operator. If x is an event, and P is a process, then $(x \rightarrow P)$, pronounced "x then P," represents a process that engages in event x and then behaves like process P. For a process P, the notation αP describes the "alphabet" of P. It is a set of all of the events that P is ever willing to engage in. For example, $\alpha\, VMS = \{in25, choc\}$.

For any CSP process P, we can discuss a *trace* of events that P may accept. For the VMS example, various traces include:

$$\begin{aligned}
&\langle\rangle \\
&\langle in25 \rangle \\
&\langle in25, choc \rangle \\
&\langle in25, choc, in25, choc, in25 \rangle
\end{aligned}$$

For two traces tr and tr', define $tr \frown tr'$ to be their concatenation.

The next important operator is "choice," denoted by "$|$". If x and y are distinct events, then:

$$(x \rightarrow P \mid y \rightarrow Q)$$

denotes a process that accepts x and then behaves like P or accepts y and then behaves like Q. For example, a new vending machine can accept either a coin and output a chocolate, or accept a bill and output an ice cream cone:

$$VMS2 = (bill \rightarrow cone \rightarrow VMS2 \mid in25 \rightarrow choc \rightarrow VMS2)$$

CSP offers a more general choice function (for choosing between many inputs succinctly), but the Flume model only requires simple choice.

A related operator is *internal (nondeterministic) choice*, is denoted "$\sqcap$". In simple choice, the machine reacts exactly to events it fields from the machine's user. In nondeterministic choice, the machine behaves unpredictably from the perspective of the user, maybe because the machine's description is underspecified, or maybe because the machine is picking from a random number generator. For instance, a change machine might return coins in any order, depending on how the machine was last serviced:

$$\begin{aligned}
CHNG = (in25 \rightarrow (&out10 \rightarrow out10 \rightarrow out5 \rightarrow CHNG \sqcap \\
&out10 \rightarrow out5 \rightarrow out10 \rightarrow CHNG))
\end{aligned}$$

That is, the machine takes as input a quarter, and returns two dimes and a nickel in one of two orderings. Another standard operator, "external choice" denoted "$\Box$", has different semantics but does not appear in Flume's model.

CSP provides useful predefined processes like $STOP$, the process that accepts no events, and $SKIP$, the process that shows a successful termination and then behaves like $STOP$. Other processes like DIV, RUN and $CHAOS$ are standard in the literature, but are not required here.

The next class of operators relate to parallelism. The notation:

$$P \underset{A}{\|} Q$$

denotes P running in parallel with Q, synchronizing on events in A.[8] This means a stream of incoming events can be arbitrarily assigned to either P or Q, assuming those events are not in A. However, for events

[8] Parallelism differs between Hoare's original CSP formulation and more modern formulations, like Schneider's. We use Schneider's "interface parallelism" in this model.

in A, both P and Q must accept them in synchrony. As an example, consider the vending machine and the change machine running in parallel, synchronizing on the event *in25*:

$$FREELUNCH = VMS \underset{\{in25\}}{\parallel} CHNG$$

Possible traces for this new process are the various interleavings of the traces for the two component machines that agree on the event *in25*. For instance:

$$\langle in25, choc, out10, out10, out5, \ldots\rangle$$
$$\langle in25, out10, choc, out10, out5, \ldots\rangle$$
$$\langle in25, out10, out10, choc, out5, \ldots\rangle$$
$$\langle in25, choc, out10, out5, out10, \ldots\rangle$$
$$\langle in25, out10, out5, out10, choc, \ldots\rangle$$

are possible execution paths for *FREELUNCH*.

Another variation on parallel composition is arbitrary interleaving, denoted: $P \,|||\, Q$. In interleaving, P and Q never synchronize, operating independently of one another. $P \,|||\, Q$ is therefore equivalent to $P \parallel_{\{\}} Q$, which means P and Q run in parallel and synchronize on the empty set.

Processes that run in parallel can communicate with one another over *channels*. A typical channel c can carry various values v, denoted $c.v$.[9] This is represented as the sending process accepting the event $c!v$ while the receiving process accepts the event $c?x$ (where x is thus far unbound) and sets x to v. Communication on a channel is possible only when the sender and receiver processes are in the respective states simultaneously. If one process is at the suitable state and the other is not, the ready process waits until its partner becomes ready. In a slight deviation from Hoare's semantics, channels here are bidirectional: messages can travel independently in either direction across a channel. The Flume model uses channels extensively.

The next important CSP feature is *concealment* or *hiding*. For a process P and a set of symbols C, the process $P\backslash C$ is P with symbols in C hidden or concealed. The events in C become internal transitions, that cannot be observed by other processes through synchronization or channel communication. Concealment can induce *divergence*—an infinite sequence of internal transitions. For instance, the process $P = (c \rightarrow P)\backslash\{c\}$ diverges immediately, never to be useful again. The use of concealment in the Flume model is careful never to induce divergence in this manner.

Concealment enables *subroutines* (or *subordination*, in Hoare's terminology). For two process P and Q whose alphabets fulfill $\alpha P \subseteq \alpha Q$, the new process $P \mathbin{/\!/} Q$ is defined as $(P \parallel Q)\backslash\alpha P$. This means that the subroutine P is available within Q, but not visible to the outside world. The notation $p{:}P \mathbin{/\!/} Q$ means a particular instance p of the subroutine P is available in Q. Then an event such as $p!x?y$ within Q means that Q is calling subroutine p with argument x, and that the return value is placed into y. Within P, the event $?x$ means receive the argument x from the caller, and the event $!y$ means return the result y to the caller.

A final important language feature is *renaming*. Given a "template" process P, the notation $i{:}P$ means a "renaming" of P with all events prefixed by i. That is, if the event $c!v$ appears in P, then the event $i.c!v$ appears in $i{:}P$, where $i.c$ is the channel c that has been renamed to $i.c$. Thus, for any $i \neq j$, the alphabets of $i{:}P$ and $j{:}P$ are disjoint: $\alpha(i{:}P) \cap \alpha(j{:}P) = \{\}$. This concludes our whirlwind tour of CSP features. We refer the reader to Hoare's [16], Schneider's [20] and Roscoe's [19] books for many more details.

9 If the channel has a compound name like $i.c$, its values are respectively denoted $i.c.v$. Channel names are prefix-free so this is never ambiguous.

The Stable Failures Model We now expand upon the idea of traces to develop an idea of process equivalence in CSP. The traces of P (denoted $traces(P)$) is the set of all traces accepted by the process P. The refusals of P (denoted $refusals(P)$) is a set of sets. A set X is in $refusals(P)$ if and only if P deadlocks when offered any event from X. For instance, consider the process P_0:

$$P_0 = (a \rightarrow STOP \sqcap b \rightarrow STOP) \quad .$$

We write that $refusals(P_0) = \{\{a\}, \{b\}\}$. That is, P_0 can nondeterministically choose the left branch, in which case it will only accept $\{a\}$ and will refuse $\{b\}$. On the other hand, if it nondeterministically chooses the left branch, it will accept $\{b\}$ and refuse $\{a\}$. Thus, due to nondeterminism, we write $refusals(P)$ as above, and *not* as the flattened union $\{a, b\}$.

The notation $Q \downarrow$ is a predicate that denotes the process Q is "stable." Unstable states are those that transition internally, or those that diverge. For example, the process P_0 begins at an unstable state, since it can make progress in either the left or right direction without accepting any input. However, once it makes its first internal transition, arriving at either $a \rightarrow STOP$ or $b \rightarrow STOP$, it becomes stable. A process that diverges, such as $(c \rightarrow P)\backslash\{c\}$, has no stable states.

Though the CSP literature explores many notions of process equivalence, this paper uses the "stable failures" model, given in Hoare's book [16] and rephrased by Schneider [20] and Roscoe [19]. For a process P, the *stable failures* of P, written $\mathcal{SF}[\![P]\!]$, are defined as:

$$\begin{aligned}\mathcal{SF}[\![P]\!] = \{(s, X) \mid s \in traces(P) \ \wedge \ P/s\downarrow \ \wedge \\ X \in refusals(P/s)\}.\end{aligned}$$

In other words, the failures of P captures which traces P accepts, and which sets it refuses after accepting those traces. For example:

$$\mathcal{SF}[\![P_0]\!] = \{(\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\} \quad .$$

In the stable failures model, two processes P and Q are deemed equivalent if and only if $\mathcal{SF}[\![P]\!] = \mathcal{SF}[\![Q]\!]$.

Lastly, CSP offers a way to identify processes in states other than their initial states: the process P/tr is P advanced to the state after the trace tr has occurred. Next, we often talk about the effects of "purging" certain events from traces and process states. The operator "$\upharpoonright$" denotes *projection*. The trace $tr \upharpoonright A$ is the trace tr projected onto the set A, meaning all events not in A are removed. For instance, if $A = \{a\}$, and $tr = \langle a, b, c, d, a, b, c\rangle$, then $tr \upharpoonright A = \langle b, c, d, b, c\rangle$. For a set C, the set $C \upharpoonright A$ is simply the intersection of the two. Define two projected processes $P \upharpoonright A$ and $Q \upharpoonright A$ equivalent if and only if $\mathcal{SF}[\![P]\!] \upharpoonright A = \mathcal{SF}[\![Q]\!] \upharpoonright A$, where:

$$\mathcal{SF}[\![P]\!] \upharpoonright A = \{(tr \upharpoonright A, X \cap A) \mid (tr, X) \in \mathcal{SF}[\![P]\!]\}$$

and similarly for Q.

.2 Per-process Queues (*QUEUES*)

Each kernel process $i{:}K$ needs it own set of queues, to handle messages received asynchronously from other processes. For convenience, we package up all of the queues in a single process $i{:}QUEUES$, which $i{:}K$ can access in all of its various states. The channel q serves communication between the queues and the kernel process. The building block of this process is a single $QUEUE$ process, similar to that defined in Hoare's book. This process is parameterized by the value stored in the queue, and of course the queue starts out empty:

$$QUEUE = QUEUE_{<>}$$

From here, we define state transitions:

$$
\begin{aligned}
QUEUE_{\langle\rangle} &= \big(?(\mathsf{enqueue}, x) \;\rightarrow\; QUEUE_{\langle x\rangle} \mid \\
&\qquad ?(\mathsf{select}, j)!\{\} \;\rightarrow\; QUEUE_{\langle\rangle}\big) \\
QUEUE_{\langle x\rangle^\frown s} &= \big(?(\mathsf{enqueue}, y) \;\rightarrow\; \mathsf{if}\ \#s + 1 < N_Q \\
&\qquad\qquad \mathsf{then}\ QUEUE_{\langle x\rangle^\frown s^\frown \langle y\rangle} \\
&\qquad\qquad \mathsf{else}\ QUEUE_{\langle x\rangle^\frown s} \mid \\
&\qquad ?(\mathsf{dequeue})!x \;\rightarrow\; QUEUE_s \mid \\
&\qquad ?(\mathsf{select}, j)!\{j\} \;\rightarrow\; QUEUE_{\langle x\rangle^\frown s}\big)
\end{aligned}
$$

Note that these queues are bounded beneath N_Q elements. Attempts to enqueue messages on filled queues result in dropped messages. The model combines many *QUEUE* subprocesses into a collection processes called *QUEUESET*:

$$QUEUESET = \|_{i\in\mathcal{P}}\ i{:}QUEUE$$

The process called *QUEUES* communicates with kernel processes. Recall that $i.q$ is the channel shared between $i{:}K$ and $i{:}QUEUES$:

$$
\begin{aligned}
QUEUES = s &: QUEUESET \mathbin{/\!/} sel : QSELECT_s \mathbin{/\!/} \mu X \bullet \\
&\big(q?(\mathsf{enqueue}, j, m) \;\rightarrow\; s.j!(\mathsf{enqueue}, m) \\
&\quad \rightarrow\; X \mid \\
&\ q?(\mathsf{dequeue}, j) \;\rightarrow\; s.j!(\mathsf{dequeue})?m \\
&\quad \rightarrow\; q!m \;\rightarrow\; X \mid \\
&\ q?(\mathsf{select}, Y) \;\rightarrow\; sel!Y?Z \;\rightarrow\; q!Z \;\rightarrow\; X \mid \\
&\ q?(\mathsf{clear}) \;\rightarrow\; QUEUES\big)
\end{aligned}
$$

Finally, the point of *QSELECT* is to determine which of the supplied queues have pending messages. This process uses tail recursion to add to the variable Z as readied queues are found.

$$
\begin{aligned}
QSELECT_s = Z &: VAR \mathbin{/\!/} ?Y \;\rightarrow \\
&Z := \{\}\,; \\
&\big(\mu X \bullet (\mathbf{if}\ Y = \{\} \\
&\qquad \mathbf{then}\ (!Z \;\rightarrow\; QSELECT_s) \\
&\qquad \mathbf{else\ pick}\ j \in Y\,; \\
&\qquad\quad Y := Y - \{j\}\,; \\
&\qquad\quad (s.j!(\mathsf{select}, j) \rightarrow s.j?A \;\rightarrow \\
&\qquad\quad (Z := Z \cup A\,;\ X))\big)
\end{aligned}
$$

.3 Determining Tag Sizes

We can solve for how big α must be in terms of β and ϵ. *Partitioning* requires that functions from G must be injective, giving $2^{4\beta} \leq 2^{\alpha}$, or equivalently, $\alpha \geq 4\beta$. As for *unpredictability*, g is chosen randomly from G, so it will output elements in $\{0,1\}^{\alpha}$ in random order. After 2^{β} calls, g outputs elements from a set sized

$2^\alpha - 2^\beta$ at random. Since $\alpha \geq 4\beta 1$, this "restricted" range for g still has well in excess of $2^{\alpha-1}$ elements. Failure occurs when a process can predict the output of g, which happens with probability no greater than $2^{\alpha-1}$. Thus, $\alpha - 1 \geq \epsilon$. Combining these two restrictions, $\alpha \geq \max(\epsilon + 1, 4\beta)$. For $\beta = 80$ and $\epsilon = 100$ we get $\alpha = 320$.

.4 Proof

Alphabets The relevant high, mid and low alphabets were defined in Section 5.1. The rest of the events in the Flume model (like communication through the switch, to the process or tag manager, etc.) are all hidden by the CSP-hiding operators, as given in Section 4.7. For convenience, we define the set of events that correspond to kernel process i's incoming system calls, and a set of event that correspond to process i's responses:

$$\begin{aligned}
C_i \triangleq & \big\{ i.s.(S, I, \mathsf{create_tag}, w) \mid \\
& \quad S, I \subset \mathcal{T} \wedge w \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{None}\} \big\} \cup \\
& \big\{ i.s.(S, I, \mathsf{change_label}, w, L) \mid \\
& \quad S, I, L \subset \mathcal{T} \wedge w \in \{\mathsf{Integrity}, \mathsf{Secrecy}\} \big\} \cup \\
& \big\{ i.s.(S, I, \mathsf{get_label}, w) \mid \\
& \quad w \in \{\mathsf{Integrity}, \mathsf{Secrecy}\} \big\} \cup \cdots
\end{aligned}$$

and so on for all system calls. The returns from system calls are:

$$\begin{aligned}
R_i \triangleq & \big\{ i.s.(S, I, t) \mid S, I \subset \mathcal{T} \wedge t \in \mathcal{T} \big\} \cup \\
& \big\{ i.s.(S, I, r) \mid S, I \subset \mathcal{T} \wedge r \in \{\mathsf{Ok}, \mathsf{Error}\} \big\} \cup \\
& \big\{ i.s.(S, I, L) \mid S, I, L \subset \mathcal{T} \big\} \cup \\
& \big\{ i.s.(S, I, O) \mid S, I \subset \mathcal{T} \wedge O \subset \mathcal{O} \big\} \cup \\
& \big\{ i.s.(S, I, p) \mid S, I \subset \mathcal{T} \wedge p \in \mathcal{P} \big\}
\end{aligned}$$

Also, we often describe the failures of a process P projected onto the low alphabet $LO_t - MID_t$, abbreviated:

$$\mathcal{L}_t[\![P]\!] \triangleq \mathcal{SF}[\![P]\!] \upharpoonright (LO_t - MID_t)$$

Proof Consider any two traces tr and tr' that are equivalent when projected to $LO_t \cup MID_t$. Perform induction over the length of the traces tr and tr'. To do so, invent a function $\lambda(\cdot)$:

$$\lambda(tr) \triangleq \#\,(tr \upharpoonright (LO_t \cup MID_t))$$

that outputs the number of low and mid events in a trace. It immediately follows that $\lambda(tr) = \lambda(tr')$. We first show the theorem holds for all traces tr and tr' such that $\lambda(tr) = \lambda(tr') = 0$. We then assume it holds for all traces with $\lambda(tr) = \lambda(tr') = k - 1$ and prove it holds for all traces with $\lambda(tr) = \lambda(tr') = k$.

Base Case For the base case, consider all $tr, tr' \in traces(SYS)$ such that $\lambda(tr) = \lambda(tr') = 0$. In other words, $tr, tr' \in (HI_t - MID_t)^*$.

At the system startup (SYS after no transitions), all of the kernel process $i{:}K$ are waiting on a message of the form $i.b$ before they spring to life. Until such a message arrives, $i{:}K$ will refuse all events C_i and R_i. The one exception is the process init, which is already waiting to accept incoming system calls when the system starts. By construction $S_{\mathsf{init}} = \{\}$ and $I_{\mathsf{init}} = \mathcal{T}$. Since $t \notin S_{\mathsf{init}}$, $C_{\mathsf{init}} \cup R_{\mathsf{init}} \subseteq LO_t$. Therefore, the system refuses all high events at startup, and $tr = \langle\rangle$ is the only trace of SYS without low symbols (and for which $\lambda(tr) = 0$). For $tr = tr' = \langle\rangle$, the claim trivially holds.

Inductive Step, $l \notin MID_t$ For the inductive step, assume the claim holds for all traces tr, tr' of SYS such that $tr \upharpoonright (LO_t \cup MID_t) = tr' \upharpoonright (LO_t \cup MID_t)$ and also $\lambda(tr) = \lambda(tr') = k - 1$. Now, we seek to show the claim holds for all equivalent traces with one more low symbol.

Given an arbitrary trace $tr \in traces(SYS)$ such that $\lambda(tr) = k$, write tr in the form $tr = p \frown l \frown h$, where p is a prefix of tr, $l \in LO_t \cup MID_t$ is a single low event, and $h \in (HI_t - MID_t)^*$ are traces of high events. There are two cases : $l \notin MID_t$ and $l \in MID_t$. Consider the first here ($l \notin MID_t$), and see Section .4 below for the second.

Write the right trace in the same form: $tr' = p' \frown l \frown h'$. It suffices to show that $\mathcal{L}_t[\![S/tr]\!] = \mathcal{L}_t[\![S/(p \frown l)]\!]$. Indeed, if we have shown this equality for arbitrary tr, then the same applies for S/tr', meaning $\mathcal{L}_t[\![S/tr']\!] = \mathcal{L}_t[\![S/(p' \frown l)]\!]$. By inductive hypothesis, $\mathcal{L}_t[\![S/p]\!] = \mathcal{L}_t[\![S/p']\!]$, and therefore $\mathcal{L}_t[\![S/(p \frown l)]\!] = \mathcal{L}_t[\![S/(p' \frown l)]\!]$. By transitivity, we have that $\mathcal{L}_t[\![S/tr]\!] = \mathcal{L}_t[\![S/tr']\!]$, which is what needs to be proven. Thus, the crux of the argument is to show that the high events of tr do not affect low's view of the system; the second trace tr' is immaterial.

We consider the event l case-by-case over the relevant events in SYS:

- $l \in R_i$ for some i

  That is, l is a return from a system call into user space. Because l is a low event, l is of the form $i.s.(S, I, O, \dots)$ where $t \notin S$. After this event, $i{:}K$ is in a state ready to receive a new system call ($i : K_{S,I,O}$). Because all events in h are high events, none are system calls of the form $i.s.(S, I, O, \dots)$ with $t \notin S$, and therefore, none can force $i{:}K$ into a different state. In other words, the events h can happen either before or after l; SYS will accept (and refuse) the same events after either ordering. That is:

  $$\mathcal{L}_t[\![SYS/(p \frown l \frown h)]\!] = \mathcal{L}_t[\![SYS/(p \frown h \frown l)]\!].$$

  We can apply the inductive hypothesis to deduce that:

  $$\mathcal{L}_t[\![SYS/(p \frown h)]\!] = \mathcal{L}_t[\![SYS/p]\!]$$

  Appending the same event l to the tail of each trace gives:

  $$\mathcal{L}_t[\![SYS/(p \frown h \frown l)]\!] = \mathcal{L}_t[\![SYS/(p \frown l)]\!]$$

  and by transitivity:

  $$\mathcal{L}_t[\![SYS/(p \frown l \frown h)]\!] = \mathcal{L}_t[\![SYS/(p \frown l)]\!]$$

  which proves the claim for this case.

  Note two special events here: first, a return from $\mathsf{create\_tag}$ in which i receives the special tag t; and second, a return from recv in which i receives the capability t^- from another process. After either system call return, $i{:}K_{S,I,O}$ transitions to some new state $i{:}K_{S,I,O \cup \{t^-\}}$. From this point forward, the failures of $i{:}K$ lie outside of $LO_t - MID_t$, and need not be considered by our extended definition of noninterference.
- $l = i.s.(S, I, O, \mathsf{create\_tag}, w)$ for some $i \in \mathcal{P}$, and some $w \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{None}\}$

  After accepting this event, the process $i{:}K$ can no longer accept system calls; it can only accept a response in the form $i.s.(S, I, O, t')$ for some tag t'. Since $l \in LO_t$, it follows that $t \notin S$ for both the system call and its eventual reply. The high events in h could affect the return value to this system call (and therefore $\mathcal{SF}[\![S/tr]\!]$) if the space of t's returned somehow depends on h, because h changed the state of the shared tag manager. An inspection of the tag manager shows that its state only changes as a result of a call to $e = j.g.(S', I', O', \mathsf{create\_tag}, w)$ for some process j, and labels S', I' and O'. Such a call would result in a tag such as $t' = g(S', I', O', x)$ being allocated, for some arbitrary x. Because $e \in h$ is a high event, $t \in S'$. Because l is a low event, $t \notin S$. Thus, $S' \neq S$, and assuming g is injective, it follows that $t' \neq t$, for all x. Therefore, events in h cannot influence which tags t' might be allocated as a result of a call to $\mathsf{create\_tag}$. We apply the same argument as above, that h and l can happen either before or after one another without changing the failures of the system.

– $l = i.s.(S, I, O, \mathsf{change\_label}, w, L)$ for some $i \in \mathcal{P}$, $w \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{None}\}$ and $L \subseteq \mathcal{T}$.
  Consider all traces of the form $tr = p \frown l \frown h$, where l is as above. As usual, we consider all elements of h that might affect process $i{:}K$ after it has accepted event l. After l, the process $i{:}K$ can only accept an event of the form $i.s.(S, I, O, r)$ for $r \in \{\mathsf{Ok}, \mathsf{Error}\}$, to indicate whether the label change succeeded or failed.
  The only way an event in h can influence the outcome r is to alter the composition of $\hat{O}$, which the tag manager checks on i's behalf by answering $i.g.(\mathsf{check+})$ and $i.g.(\mathsf{check\text{-}})$ within the CHECK subprocess. That is, some high process j must request a new tag with system call $e = j.s.(S', I', O', \mathsf{create\_tag}, w)$, where $t \in S'$, and $e \in h$. Moreover, the kernel must return as a result some new tag t' such that $t' \in L$, where L is the label that low process i desires to change to. If both of these conditions hold, then the tag manager might have switched to a new state in which $t'^{+} \in \hat{O}$ or $t'^{-} \in \hat{O}$, meaning the kernel's response to event l could have changed based on h.
  Such an h is extremely unlikely. Since l happened *before* h in trace tr, t' can only be a member of L if U_i "predicted" the output of the tag manager, which it can do with negligible probability ($2^{-\epsilon}$). With high probability, h does not contain any elements that can affect r or future states of $i{:}K$ after event l. Thus, $\mathcal{L}_t[\![p \frown l \frown h]\!] = \mathcal{L}_t[\![p \frown l]\!]$, which proves the case.

– $l = i.s.(S, I, O, \mathsf{get\_label}, w)$ for some w.
  This call only outputs information about what state a kernel process is in; this state only updates as a result of low events $i.s.(S, I, O, \mathsf{change\_label})$.

– $l = i.s.(S, I, O, \mathsf{get\_caps})$.
  There are three state transitions that can alter the reply to the $\mathsf{get\_caps}$ system call: $i.s.(S, I, O, \mathsf{create\_tag}, w)$, $i.s.(S, I, O, \mathsf{drop\_caps}, L)$ or $i.s.(S, I, O, \mathsf{recv}, j)$. None of these calls are equal to an event in h, since they are low events and h contains only high events.

– $l = i.s.(S, I, \mathsf{fork})$
  The only event $i{:}K$ will accept after l is $i.s.(S, I, O, k)$ where k is the process ID of the newly-forked child. By definition of CHOOSE above, there exists some x such that $k = g(s(S), s(I), s(O), x)$. If an event $e \in h$ causes a process ID to be chosen, it would be of the form $p = g(s(S'), s(I'), s(O'), y)$, for some y, and some S' such that $t \in S'$. That l is a low symbol implies that $t \notin S$ and $S \neq S'$. If g is injective then $k \neq p$. Therefore, event e will never change the value k that this kernel process might output next as its reply to the system call l.
  The other result of the fork system call is that now, a new process k is running. That is, $k : K$ has moved out of the "birth state" and is now willing to accept incoming system calls in state $(k : K_{S,I,O})$. The same arguments as above apply here. Because k was forked by a low process, it too is a low process, expecting only low symbols before it transitions to a new state. Therefore, the events in h cannot affect its state machine.

– $l = i.s.(S, I, \mathsf{send}, j, X, m)$ for some j, X, m.
  The outcome of the send operation depends only on whether $X \subseteq O$ or not. It therefore does not depend on h.

– $l = i.s.(S, I, O, \mathsf{recv}, j)$
  The event after l that $i{:}K$ accepts is $i.s.(S, I, O, m)$ for some message m. It might also change to a different state if the process j sent capabilities. The relevant possibility for $e \in h$ to consider is $e = j.s.(S', I', O', \mathsf{send}, i, \{t'^{+}\}, \langle\rangle)$, for some high process j with $t \in S'$. The claim is that this message will never be enqueued at i and therefore will not affect i's next visible event. Say that process j has ownership O' and dual privileges D'. That t is an export-protection tags implies $t^{-} \notin \hat{O}$; that all events in h are not in MID_t implies $t^{-} \notin O'$; that $l \notin \mathit{MID}_t$ implies $t^{-} \notin O$. Thus, $t \notin D \cup D'$. Also, because

i is a low process $t \notin S$. Therefore, $t \in S' - D'$ and $t \notin S \cup D$, which implies that $S' - D' \not\subseteq S \cup D$, and the kernel will not enqueue or deliver j's message to i. Again, we have that h does not affect i's possibilities for the next message it receives.

The remaining events (e.g., drop_caps, exit, etc.) follow similarly and are elided for brevity.

Inductive Step, $l \in MID_t$ The previous section considered all traces of the form $tr = p \frown l \frown h$ where $l \notin MID_t$; it remains to cover the case of $l \in MID_t$. When l consists of any system call aside from send or drop_caps, we claim that $\mathcal{L}_t[\![SYS/p \frown l]\!] = \mathcal{L}_t[\![SYS/p]\!]$. That is, on either side of such an event, all processes in a MID_t state stay in that state, and don't communicate with any low processes. If such processes allocate new tags or process IDs (via fork), *CHOOSE*'s partitioning of the identifier space prevents declassifiers from taking tag or process IDs from low processes.

If l is an event of the form $i.s.(S, I, O, \mathsf{send}, j, X, m)$ where j corresponds to a "low" process, then l can affect the failures of a low process. However, we can apply the same argument as above to conclude that the elements of h do not interfere with j's receipt of i's message, or its subsequent states. If $t^- \in X$, then process j becomes a declassifier upon receipt of i's message, its traces no longer in the set $LO_t - MID_t$. High events (like those in h) can then influence i, but the definition no longer considers j's failures.

If l is an event of the form $i.s.(S, I, O, \mathsf{drop\_caps}, X)$, and $t^- \in X$, then l represents process i's rescinding its status as a declassifier; it becomes a low process like any other. No high event can interfere with the drop_caps operation, since its success depends only on O and X. Once the capability is dropped, i can be analyzed under the first set of cases (pertaining to events in LO_t but not in MID_t).

We have covered all of the relevant cases, and the theorem follows by induction. ∎

.5 select and Timing

Consider a new system call, select, that involves an explicit timeout:

- $Y \leftarrow \mathsf{select}(t, X)$
  Given a set of process indices X, return a set $Y \subseteq X$ such that all $j \in Y$, calling $\mathsf{recv}(j)$ will yield immediate results. This call will block until Y is non-empty, or until t clock ticks expire.

A new kernel subprocess *SELECT* handles the new system call; it allows a user program to wait for the first available receive channel to become readable:

$$
\begin{aligned}
SELECT_{S,I,O} = \big(s \underset{\kappa}{?} (\mathsf{select}, t, A) \ \rightarrow \\
&\big(\mu X \bullet (q!(\mathsf{select}, A) \rightarrow q?B \ \rightarrow \\
&\qquad \textbf{if } B = \{\} \\
&\qquad \textbf{then } INTRECV{*}_{S,I,O} \, ; X \\
&\qquad \textbf{else } s \underset{\kappa}{!} B \ \rightarrow \ K_{S,I,O}) \\
&\triangle_t \ \big(s \underset{\kappa}{!} \{\} \ \rightarrow \ K_{S,I,O}\big)\big)
\end{aligned}
$$

There are three new CSP operators here. The first is $\mu X \bullet F(X)$, which (following Hoare's original syntax) is the recursion operator: the process X such that $X = F(X)$. The syntax $P;Q$ denotes the process P followed by Q upon P's successful termination. Successful termination is denoted by the special CSP process *SKIP*. Lastly, the "timed interrupt operator" $\triangle_t$ [20] interrupts the selection process after t clicks of the clock and outputs an empty result set.

In this formulation, the process *SELECT* calls subprocess *INTRECV**, which behaves mostly like *INTRECV*, except it keeps receiving until an admissible message arrives:

$$\begin{aligned} \mathit{INTRECV}^*_{S,I,O} \; = \; & c?(S_{\text{in}}, I_{\text{in}}, j, X, m) \; \rightarrow \\ & g!(\mathsf{dual\_privs}, O) \rightarrow g?D \rightarrow \\ & \textbf{if } (S_{\text{in}} \subseteq S \cup D) \; \wedge \; (I - D \subseteq I_{\text{in}}) \\ & \textbf{then } q!(\mathsf{enqueue}, (X, m)) \rightarrow \mathit{SKIP} \\ & \textbf{else } \mathit{INTRECV}^*_{S,I,O} \end{aligned}$$

With the inclusion of the select operation, the Flume CSP model now explicitly models time. We must update our definitions and proof accordingly. Schneider develops a full notion of process equivalence in timed CSP [20], but the mechanics are complex. Instead, we suggest a technique introduced by Ouaknine [41] and also covered by Schneider [20]: convert a timed model into an untimed model with the introduction of the event *tock*, which represents a discrete unit of time's passage. In particular, Schneider provides the Ψ function for mapping processes from timed CSP to discrete-event CSP with a *tock* event representing the passage of time. In the proof of noninterference, consider *tock* a low event, that is not hidden by any concealment operator. Then apply the Ψ translation to all states of the Flume model.