Noninterference for a Practical DIFC-Based Operating System

Maxwell Krohn
CyLab
Carnegie Mellon University
Pittsburgh, PA

Eran Tromer
CS and AI Laboratory
Massachusetts Institute of Technology
Cambridge, MA

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.

I. 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 ([11], [11]) have inadvertently included high-bandwidth covert channels in their very interface, allowing information to leak against intended security policies (see Section III 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 II 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 III describes potential pitfalls, motivating a formal approach. Section IV 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 V 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 classifiers. 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.

II. 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 $T$ 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 $b \in T$ may label Bob’s private data.

Labels are subsets of $T$. 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 ∈ 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 ∈ 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.

Distribution 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.

A. Capabilities
Flume represents privilege using two capabilities per tag. Capabilities are objects from the set $O = T \times \{-, +\}$. For tag t, the capabilities are denoted $t^+$ and $t^-$. Each process p owns a set of capabilities $O_p \subseteq O$. 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 data, while $t^-$ lets it remove t from its secrecy label, effectively declassifying any secret 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

$$D_p = \{ t | t^+ \in O_p \land t^- \in O_p \}$$

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 ∈ T and sets $O_p = O_p \cup \{ t^+, t^- \}$, granting p dual privilege for t. Tag allocation should not expose any information about system state.

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

$$O^+ = \{ t | t^+ \in O \}, \quad O^- = \{ t | t^- \in O \}$$

B. 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 II-D). A process p’s effective set of capabilities is given by:

$$\hat{O}_p = O_p \cup \hat{O}$$

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

$$\hat{D}_p = \{ t | t^+ \in \hat{O}_p \land t^- \in \hat{O}_p \}$$

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 $\hat{O}_p$).

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 $\hat{O}_p$ (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.

C. 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.

1) 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 [11], [13], [11], but implicit label changes turn the labels themselves into covert channels [12], [2] (see Section III). 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:

$$L' - L \subseteq (\hat{O}_p)^+ \quad \text{and} \quad L - L' \subseteq (\hat{O}_p)^-$$

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. $t^- \in O_p$). Likewise, t can be added only if $t^+ \in \hat{O}_p$.

2) 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 classifiers. 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 $D_p$, 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 $S_p - D_p$ and raise its integrity to $I_p + D_p$. It could receive the most messages if it were to raise secrecy to $S_p + D_p$ and lower integrity to $I_p - D_p$.

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

$$S_p - D_p \subseteq S_q \cup \hat{D}_q \quad \text{and} \quad I_q - \hat{D}_q \subseteq I_p \cup \hat{D}_p$$

3) 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 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
Figures 1–3). Imagine processes and avoid unwanted label changes. Also introduces "receive labels," which let processes filter out traffic floating label system like Asbestos, creates a new tag and grants t+ to the global set 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 Sq, which it can do since t+ ∈ O ⊆ Ot. 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.

D. 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+ ∈ O and t− ∉ O. For a process p to achieve such a result, it creates a new tag t and grants t+ to the global set 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 Sq, which it can do since t+ ∈ O ⊆ Ot. 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.

III. COVERT CHANNELS IN DYNAMIC LABEL SYSTEMS

As described in Section II-C, 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_1} = {}. q’s version of the secret starts out initialized to all 0s, but it will overwrite some of those bits during the attack.

Next, p communicates selected bits of the secret to its helpers. If the i-th bit of the message is equal to 0, then p sends the message “0” to the process q_i. If the i-th bit of the message is 1, p does nothing. Figure 2 shows this step. When receiving this 0 bit, q_i’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 in a 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.

IV. 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 V 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 A). 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 k_i K that behaves according to a strict state machine. 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 k_i.s between the U_i and k_i K.

1The CSP notation i.K means the i-th instance of a template process 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 process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.

The kernel halves communicate with one another to deliver IPCs initiated by user processes. Also inside the kernel, a global process (TAGMGR) manages the process ID space, and SWITCH enables all user-visible interprocess communication. Arrows denote CSP communication channels.
and the trace for the kernel half $i:K$ is of the form

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

That is, $U_i$ is sending a message (create\_tag, 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 $i.s$'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 and therefore that $U_i$ must now send system calls in the form:

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

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

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

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:

$$c!((x) \triangleq c!(S, I, O, x), \quad c?(x) \triangleq c?(S, I, O, x)$$

For example, kernel process $i:K_{(t), I, O}$ will receive $S_i.(t) \downarrow$'s call to $s?((fork)$ as expanded as $s?(\{n\}, \{n\}, \{\}, fork)$.

### D. 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 $t$.\(^2\)

$$NEWTAGS_{S,I,O} = (s?\(create\_tag, w\) \rightarrow
\quad g!(create\_tag, w)!((t, O_{new}) \rightarrow
\quad s\!t \rightarrow K_{S,I,O, O_{new}})$$

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

$$CHANGELABELS_{S,I,O} = S\cdot CHANGES_{S,I,O} \mid I\cdot CHANGES_{S,I,O}$$

$$S\cdot CHANGES_{S,I,O} = (check : CHECKS_{S,I,O} \mid
\quad s?\(change\_label, Secrecy, S'\) \rightarrow
\quad check!(S, S') \rightarrow check?r \rightarrow
\quad if \ r \ then \ s!Ok \rightarrow KS', I, O \mid
\quad else \ s!Error \rightarrow KS,I,O)$$

\(^2\) A note on notation: the channel between $U_i$ and $i:K$ is named $i.s$ as stated earlier. However, the "s:" 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 $s$.

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 $O$ (captured by a global process listening on the $g$ channel):

$$CHECKS_{S,I,O} = \{\begin{array}{l}
\quad g!(check-, \overline{L - L' - O^-}) \rightarrow g!r \rightarrow
\quad g!(check+, \overline{L' - L - O^+}) \rightarrow g!a \rightarrow
\quad (r \land a) \rightarrow CHECKS_{S,I,O}
\end{array}$$

As we will see below, the global tag register replies True to (check+, $L$) iff $L \subseteq O^+$, and replies True to (check+, $L$) iff $L \subseteq O^-$. 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 II-C).

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

$$READMYLABELS_{S,I,O} = \{\begin{array}{l}
\quad (s?\{get\_label, Secrecy\} \rightarrow s!S \rightarrow KS,I,O \mid
\quad s?\{get\_label, Integrity\} \rightarrow s!I \rightarrow KS,I,O)
\end{array}$$

A process also can discard capabilities using DROPCAPS:

$$DROPCAPS_{S,I,O} = (s?\{drop\_caps, X\} \rightarrow KS,I,O-X$$

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 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,\(^3\) but the kernel-side behavior of fork is specified as follows:

$$FORKS_{S,I,O} = (s?\{fork\} \rightarrow
\quad p!\{fork, O\} \rightarrow p!j \rightarrow
\quad s!j \rightarrow KS,I,O)$$

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:

$$GETPIDS_{S,I,O} = (s?\{getpid\} \rightarrow
\quad p!\{getpid\} \rightarrow p!i \rightarrow
\quad s!i \rightarrow KS,I,O)$$

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

$$EXITS_{S,I,O} = (s?\{exit\} \rightarrow
\quad g!(clear) \rightarrow p!(exit) \rightarrow SKIP)$$

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

\(^3\) 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.
E. 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 \(i:\text{SEND}_{S,i,O} \), 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 \( \text{SWITCH} \) via channel \( i.c \), which forwards it \( j \).

\[
\text{SEND}_{S,i,O} = (s?(send,j,X,m) \rightarrow \\
g!(\text{dual} _{privs},O) \rightarrow g?D \rightarrow \\
e!(S-D,I \cup D,j,X \cap O,m) \rightarrow K_{S,i,O})
\]

The process \( \text{SWITCH} \) 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 \( j.e!(S,I,i,X,m) \):

\[
\text{SWITCH} = [\forall i.c?(S,I,j,X,m) \rightarrow \\
(j.e!(S,I,i,X,m) \rightarrow \text{SKIP} \ || \ \text{SWITCH})]
\]

The \( \text{SWITCH} \) 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 \( \text{SWITCH} \) 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 \( \text{INTRECV} \) subprocess. After performing the label checks given by Definition 2 in Section II-C2, this process enqueues the incoming message for later retrieval:

\[
\text{INTRECV}_{S,i,O} = e!(S_{in},I_{in},j,X,m) \rightarrow \\
g!(\text{dual} _{privs},O) \rightarrow g?D \rightarrow \\
\text{if } (S_{in} \subseteq S \cup D) \land (I \setminus D \subseteq I_{in}) \text{ then } \\
g!(\text{enqueue}, (X,m)) \rightarrow K_{S,i,O} \\
\text{else } K_{S,i,O}
\]

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.

\[
\text{RECV}_{S,i,O} = (s?\text{recv},j) \rightarrow \\
g!(\text{dequeue},j) \rightarrow g!(X,m) \rightarrow \\
s!m \rightarrow K_{S,i,O,X}
\]

F. 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: \( \text{TAGMGR} \), which manages all global tag allocation and global capabilities; \( \text{QUEUES} \), which manages receive message queues, one per process; and finally \( \text{PROCMGR} \), which manages process creation, deletion, etc.

1) The Tag Manager (\( \text{TAGMGR} \)): The tag manager keeps track of a global universe of tags \( (T) \), and the global set of privileges available to all processes \((O)\). 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 \( \text{TAGMGR}_{O,\hat{T}} \). Initially, \( O \) and \( \hat{T} \) are empty:

\[
\text{TAGMGR} = \text{TAGMGR}_{\emptyset,\emptyset}(\cdot,\cdot)
\]

Once active, the tag manager services several calls:

\[
\text{TAGMGR}_{O,\hat{T}} = \text{NEWTAG}+_{O,\hat{T}} | \\
\text{NEWTAG}−_{O,\hat{T}} | \\
\text{NEWTAG}_0 +_{\hat{T}} | \\
\text{DUALPRIVS}_{O,\hat{T}} | \\
\text{CHECK}+_{O,\hat{T}} | \\
\text{CHECK}−_{O,\hat{T}}
\]

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 \( Y \):

\[
\text{CHOOSE}_Y = ?(S,I,O) \rightarrow \bigcap_{y \in Y} (\forall y \rightarrow \text{STOP})
\]

That is, the subprocess \( \text{CHOOSE} \) nondeterministically picks an element \( y \) from \( Y \) and returns it to the caller. As we will see in Section V, \( \text{CHOOSE} \)'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:

\[
\text{NEWTAG}+_{O,\hat{T}} = \text{choose:CHOOSE}−_{O,\hat{T}} / \\
[\forall i.g?(\text{create} _{tag}, \text{Add}) \rightarrow \\
\text{choose}!(S,I,O)?t \rightarrow \\
i.g!(t,\{t^+\}) \rightarrow \\
\text{TAGMGR}_{O \cup \{t^+\},\hat{T} \cup \{t\}}]
\]

That is, the subprocess \( \text{NEWTAG}^+ \) picks a channel \( i \) such that \( i.g \) has input available. Then, it chooses an unallocated tag \( t \) via \( \text{CHOOSE} \) 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 \( \text{NEWTAG}−_O \) and \( \text{NEWTAGO} \) similarly:

\[
\text{NEWTAG}−_O = \text{choose:CHOOSE}−_{O,\hat{T}} / \\
[\forall i.g?(\text{create} _{tag}, \text{Remove}) \rightarrow \\
\text{choose}!(S,I,O)?t \rightarrow \\
i.g!(t,\{t^+\}) \rightarrow \\
\text{TAGMGR}_{O \cup \{t^+\},\hat{T} \cup \{t\}}]
\]

\[
\text{NEWTAG}0−_O = \text{choose:CHOOSE}−_{O,\hat{T}} / \\
[\forall i.g?(\text{create} _{tag}, \text{None}) \rightarrow \\
\text{choose}!(S,I,O)?t \rightarrow \\
i.g!(t,\{t^+\}) \rightarrow \\
\text{TAGMGR}_{O \cup \{t^+\},\hat{T} \cup \{t\}}]
\]

The \( \text{DUALPRIVS} \) subprocess gives a user process \( U_i \) access to the shared capabilities in \( O \). On input \( O_i \), it returns \( D_i \), the set of tags for which process \( i \) has dual privilege. \( \text{DUALPRIVS} \) 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:

\[
\text{DUALPRIVS}_{\hat{O}, \hat{T}} = |v_{i}. (i.g?((\text{dual_privs}, O_i)) \rightarrow i.g!((O_i^+ \cup \hat{O}^+) \cap (O_i^- \cup \hat{O}^-)) \rightarrow \text{TAGMGR}_{\hat{O}, \hat{T}})
\]

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:

\[
\text{CHECK+}_{\hat{O}, \hat{T}} = |v_{i}. (i.g?((\text{check+}, L)) \rightarrow \text{TAGMGR}_{\hat{O}, \hat{T}})
\]

And similarly:

\[
\text{CHECK-}_{\hat{O}, \hat{T}} = |v_{i}. (i.g?((\text{check-}, L)) \rightarrow \text{TAGMGR}_{\hat{O}, \hat{T}})
\]

2) 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, \( P \). The process manager keeps track of subset \( \hat{P} \subseteq P \) to account for which of those processes identifiers have already been used. In then allocates from \( P - \hat{P} \).

\[
\text{PROCMGR}_{\hat{P}} = \text{PM-FORK}_{\hat{P}} \mid \text{PM-GETPID}_{\hat{P}} \mid \text{PM-EXIT}_{\hat{P}}
\]

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 \( j.bl((S, I, O))\), and returns child’s process ID to the parent:

\[
\text{PM-FORK}_{\hat{P}} = \text{choose : \text{CHOOSE}_{\hat{P} - \hat{P}}} \mid |v_{i}. (i.p?((S, I, O, \text{fork})) \rightarrow \text{choose}((S, I, O)) \rightarrow j.bl((S, I, O)) \rightarrow i.p!(j) \rightarrow \text{PROCMGR}_{\hat{P}})
\]

Trivially:

\[
\text{PM-GETPID} = |v_{i}. (i.p?((\text{getpid})!s) \rightarrow \text{PROCMGR})
\]

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

\[
\text{PM-EXIT} = |v_{i}. (i.p?((\text{exit})) \rightarrow \text{PROCMGR})
\]

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:

\[
\text{PROCMGR}_{\text{init}} = \text{init.bl}((\{\}, T, \{\}) \rightarrow \text{PROCMGR}_{\hat{P} - \{\text{init}\}})
\]

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

G. High Level System Definition

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

\[
\text{UPROCS} = \bigm| \bigm| U_j \bigm|_{j \in P}
\]

The kernel processes are:

\[
\text{KS} = \bigm| j: \left( (K \bigm| \text{SWITCH}) \setminus \alpha\text{SWITCH} \right) \bigm|_{j \in P}
\]

Adding in the helper process gives the complete kernel:

\[
\text{KERNEL} = (\text{KERNEL1} \bigm| \text{TAGMGR}) \setminus \alpha\text{TAGMGR} \bigm| \text{KERNEL2} \bigm| \text{PROCMMR0} \setminus \alpha\text{PROCMMR0}
\]

Finally:

\[
\text{SYS} = \text{UPROCS} \bigm| \text{KERNEL}
\]

This assembly of kernel process makes extensive use of the CSP hiding operator ("\( \setminus \)"). 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 \( i.s_1^{\alpha K} \) and \( i.s_2^{\alpha K} \). 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 /proc utility.

V. Noninterference

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

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 \( t \not\in S_q \). 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.

A. 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:

\[
\text{Notation: As per standard CSP, } \alpha K \text{ denotes the “alphabet” of process } K. \text{ Also, } P.c \equiv \{p.c \mid p \in P\}
\]
Fig. 5. Intransitive Noninterference. Arrows depict allowed influence. All influences are allowed except high to low.

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

\[
\forall tr, tr' \in \text{traces}(S) : \ tr | LO = tr' | LO \Rightarrow \ SF[S/tr] | LO = SF[S/tr'] | LO .
\]

We say that the process S exhibits noninterference with respect to the low alphabet LO iff \( \text{NI}_{LO}(S) \) 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 (such that \( t^+ \in \hat{O} \) and \( t^- \notin O \)), we define the high and low alphabets as follows. The high symbols emanate from a process with \( t \in S_i \):

\[
\text{Hi}_i \triangleq \{ \text{i.b.}(S, I, O, \ldots) | t \in S \} \cup \{ \text{i.s.}(S, I, O, \ldots) | t \in S \}
\]

The low symbols are the complement set:

\[
\text{Lo}_i \triangleq \{ \text{i.b.}(S, I, O, \ldots) | t \notin S \} \cup \{ \text{i.s.}(S, I, O, \ldots) | t \notin S \}
\]

1) 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.

2) 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 classifiers 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 \( \text{MD}_i \) that emanate from or are received by declassifier processes. \( \text{MD}_i \) has the property that \( \text{MD}_i \) and \( \text{Hi}_i \) need not be disjoint, nor do \( \text{MD}_i \) and \( \text{Lo}_i \):

\[
\text{MD}_i \triangleq \{ \text{i.b.}(S, I, O, \ldots) | t^- \in O \} \cup \{ \text{i.s.}(S, I, O, \ldots) | t^- \in O \}
\]

Next, consider all pairs of traces that differ only in high non-declassify elements, and are therefore equivalent when projected to \( \text{Lo}_i \cup \text{MD}_i \). 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 \( \text{LO} \) and \( \text{MD} \) contained in \( \alpha S \), the predicate \( \text{NI}_{\text{LO}}^{\text{MD}}(S) \) is true iff

\[
\forall tr, tr' \in \text{traces}(S) : \ tr | (\text{LO} \cup \text{MD}) = tr' | (\text{LO} \cup \text{MD}) \Rightarrow \ SF[S/tr] | (\text{LO} \cup \text{MD}) = SF[S/tr'] | (\text{LO} \cup \text{MD}) .
\]

Note that this extended definition is equivalent to the standard definition if \( \text{MD} = \{ \} \), i.e., if declassification is disallowed. We conjecture an “unwound” version of this definition is equivalent to Bossi et al.’s DP\_BNDC if \( \text{Hi}_i \) and \( \text{MD}_i \) are disjoint, but we defer the proof to future work.

B. Allocation of Global Identifiers

The model presented in Section IV is almost fully-specified, with an important exception: the process \( \text{CHOOSE} \):

\[
\text{CHOOSE}_V = ?(S, I) \rightarrow \bigcap_{y \in V}(y) \rightarrow \text{STOP}
\]

The “internal (nondeterministic) choice” operator (\( \bigcap \)) 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 \( \{1, 2, 3, \ldots\} \). 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 \( \text{create\_tag}(\text{Add}) \), 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 \(O\), 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 effects: a previously unallocated resource becomes observable. Thus,

\[
A \text{ reasonable value for } \beta \text{ expresses fewer than } 100\text{ operations, forking and constructing labels count as operations, the } \text{sequence } (\text{secrecy label).}
\]

A reasonable value for \(s\) is 80, meaning that no instance of the Flume system ever has an empty \(S\). The weak point here is that the process \(S\) may not have allocated. The weakness here is that the process \(S\) may not have allocated. The weakness here is that the process \(S\) may not have allocated.

Consider, as an example, this trace for the Flume system:

\[
tr = \langle i.b.(\{t\}, \{\}, \{\}),
\]

\[
i.s.(\{t\}, \{\}, \{\}, \text{fork}),
\]

\[
j.b.(\{t\}, \{\}, \{\}),
\]

\[
i.s.(\{\}, \{\}, \{\}, \{\}, \ldots)
\]

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

Projecting this trace onto the low alphabet yields the empty sequence \((tr \mid L_0 = \langle \rangle)\). 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, \(t\) could have forked off process \(j\), meaning:

\[
tr' = \langle k.b.(\{\}, \{\}, \{\}, \{\}),
\]

\[
k.s.(\{\}, \{\}, \{\}, \text{fork}),
\]

\[
j.b.(\{\}, \{\}, \{\}, \{\}),
\]

\[
k.s.(\{\}, \{\}, \{\}, \{\}, \ldots)
\]

A new process \(k\) is born, with secrecy label \(S_k = \{t\}\), and empty integrity and ownership. Thus, \(k\)’s actions fall into the \(Hi_i\) alphabet. Once \(k\) starts, it forks a new process, which the kernel randomly picks as \(l\). The child \(l\) runs with secrecy label \(S_l = \{t\}\), inheriting its parent’s secrecy label.

This contradicts the definition of noninterference in the stable failures model of process equivalence.

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

\[
\alpha \triangleq \log_2(\text{the number of tags})
\]

\[
\beta \triangleq \log_2(\text{maximum number of operations})
\]

\[
\epsilon \triangleq \log_2(\text{acceptable failure probability})
\]

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

Define a lookup table \(s()\), 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:

\[
G: ([0, 1]^\beta, [0, 1]^\beta, [0, 1]^\beta, [0, 1]^\beta) \rightarrow [0, 1]^\alpha
\]

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 C derives \(\alpha \geq \max(\epsilon + 1, 4, \beta)\), meaning \(\alpha = 320\) for our example parameters.

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

\[
\text{CHOOSE}_Y = ?(S, I, O) \rightarrow \bigcap_{y \in \mathcal{G}(S,I,O,Y)} (t_y) \rightarrow \text{STOP}
\]

where

\[
\mathcal{G}(S, I, O, Y) = \{ t \mid x \in T \land t = g(s(S), s(I), s(O), x) \land t \in Y \}.
\]

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 \(\text{CHOOSE}\) will ever output, based on the capabilities, secrecy and integrity labels of the calling process.

C. Theorem and Proof

The main theorem is as follows:

**Theorem 1** (Noninterference in Flume). For any security parameter \(\epsilon\), there exists an instantiation of \(\text{CHOOSE}\) such that: for any export-protection tag \(t\), for any Flume instance \(SYS\), \(Pr[\text{NIP}_{\text{HI}}(SYS) \geq 1 − 2^{-\epsilon}]\).

In other words, the system administrator of a Flume system first decides on a security parameter \(\epsilon\), expecting calamitous system collapse with probability of at most \(1 − 2^{-\epsilon}\). He instantiates \(\text{CHOOSE}\) with \(\epsilon\), then boots the system. The system 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., \(\text{UPROCS}\)) 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 \(\text{change}_\text{tag}\), which must consider the unlikely case that a low process guessed which tags a high process received from the kernel when calling \(\text{create}_\text{tag}\). See Appendix D for details.

VI. 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.

A. Refinement

Due to the well-known refinement paradox, the Flume model might satisfy noninterference, but implementations (i.e., refinements\(^5\)) of

\[Q\] is a refinement of \(P\) iff \(SF[Q] \subseteq SF[P]\).
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 $\cap$ or $\sqsupset$ operators: (1) the CHOOSE process; (2) the user processes $U_i$; (3) “scheduling”; and (4) the tick events in timed CSP. As for (1), the proof in Section D holds for some refinements of CHOOSE, such as the random function specified in Section V-B 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 D holds for arbitrary refinements of user processes $U_i$, as long as they communicate only through the designated system calls (see Section IV-A). In practice, we cannot hope to isolate the $U_i$s 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 IV-G. 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 E explores extending the Flume model to show an explicit passage of time: the event tick denotes one clock tick, and all parts of the model must stand aside and yield to tick. We conjecture that if $\text{tick} \in H_T$, then the proof holds for all refinements of the scheduler, and that if $\text{tick} \in L_O$, 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 $\text{tick} \in H_T$). 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 gateway 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.

B. 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 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 $\text{create} \_\text{tag}$, it cannot service $\text{create} \_\text{tag}$ until it replies to $\text{create} \_\text{tag}$. In practical implementations of this CSP model, such serializability 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.

C. 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 $\text{HMAC}_k(S, I, O, x) = \text{HMAC}_k(S', I', O', x')$, 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 $\text{HMAC}_k(S, I, O, x)$ without knowing $k$.

D. 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 $t \in O$ and $t' \notin O$. 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.

VII. 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 III. 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 III, 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].

VIII. 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, Nickoal 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


A. 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 work [16].

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

\[ \text{VMS} = \text{in}25 \rightarrow \text{choc} \rightarrow \text{VMS} \]

This vending machine waits for the event \( \text{in}25 \), which corresponds to the input of a quarter into the machine. Next, it accepts the event \( \text{choc} \), which corresponds to a chocolate falling out of the machine. Then it returns to the original state, with a recursive call to itself. The next important CSP feature is\( \dnormalsubscript{\|} \) parallelism. The notation:

\[ P \parallel Q \]

denotes \( P \) running in parallel with \( Q \), synchronizing on events in \( A \). 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 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 \( \text{in}25 \):

\[ \text{FREELUNCH} = \text{VMS} \parallel \text{CHNG} \]

Possible traces for this new process are the various interleavings of the traces for the two component machines that agree on the event \( \text{in}25 \). For instance:

\[ (\text{in}25, \text{choc}, \text{out}10, \text{out}10, \text{out}5, \ldots) \]

\[ (\text{in}25, \text{out}10, \text{choc}, \text{out}10, \text{out}5, \ldots) \]

\[ (\text{in}25, \text{out}10, \text{out}10, \text{choc}, \text{out}5, \ldots) \]

\[ (\text{in}25, \text{choc}, \text{out}10, \text{out}5, \text{out}10, \ldots) \]

\[ (\text{in}25, \text{out}10, \text{out}5, \text{out}10, \text{choc}, \ldots) \]

are possible execution paths for \( \text{FREELUNCH} \).

Another variation on parallel composition is arbitrary interleaving, denoted: \( P \circ \parallel Q \). In interleaving, \( P \) and \( Q \) never synchronize, operating independently of one another. \( P \parallel \parallel Q \) is therefore equivalent to \( P \circ \parallel Q \), 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 \( c.v \). 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 \( P |C \) 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 \( P \parallel Q \) is defined as \( \langle P \parallel Q \rangle \alpha P \). This means that the subroutine \( P \) is available within \( Q \), but not visible to the outside world. The notation \( p.P \parallel Q \) 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 \( y \).

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 \( e.v \) appears in \( P \), then the event \( i.e.v \) appears in \( i : P \), where \( i.e \) is the channel \( e \) has been renamed to \( i \). 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.

2) 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 \( \text{traces}(P) \)) is the set of all traces accepted by the process \( P \). The refusal of \( P \) (denoted \( \text{refusals}(P) \)) is a set of all \( P \) is in \( \text{refusals}(P) \) if and only if \( P \) deadlocks when offered any event from \( X \). For instance, consider the process \( P_0 \):

\[
P_0 = (a \rightarrow \text{STOP} \parallel b \rightarrow \text{STOP})
\]

We write that \( \text{refusals}(P_0) = \{\{a\}, \{b\}\} \). 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 \( \text{refusals}(P) \) 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 \( a \rightarrow \text{STOP} \) or \( b \rightarrow \text{STOP} \), 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:

\[
\mathcal{SF}[P] = \{(s,X) \mid s \in \text{traces}(P) \land P/s \downarrow \land X \in \text{refusals}(P/s)\}.
\]

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

\[
\mathcal{SF}[P_0] = \{(( ), \{a\} ),(( ), \{b\} ),((a), \{a,b\} ),((b), \{a,b\} )\}.
\]

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 \( P/\text{tr} \) is \( P \) advanced to the state after the trace \( \text{tr} \) has occurred. Next, we often talk about the effects of “purging” certain events from traces and process states. The operator “\( \parallel \)” denotes projection. The trace \( P \parallel A \) is the trace \( P \) projected onto the set \( A \), meaning all events not in \( A \) are removed. For instance, if \( A = \{a\} \), and \( tr = \{a,b,c,d\} \), then \( tr \parallel A = \{b,c,d\} \). For a set \( C \), the set \( C \parallel A \) is simply the intersection of the two.

Define two projected processes \( P \parallel A \) and \( Q \parallel A \) equivalent if and only if \( \mathcal{SF}[P] \parallel A = \mathcal{SF}[Q] \parallel A \), where:

\[
\mathcal{SF}[P] \parallel A = \{(tr \uparrow A, X \cap A) \mid (tr, X) \in \mathcal{SF}[P]\}
\]

and similarly for \( Q \).

B. Per-process Queues (QUEUES)

Each kernel process \( i : K \) needs its 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 \( i : QUES \), 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 \( QUEUES \) 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:

\[
\text{QUEUES} = \text{QUEUES}_{<}>
\]

From here, we define state transitions:

\[
\text{QUEUES}_{<} = \left(\langle \text{enqueue}, x \rangle \rightarrow \text{QUEUES}_{<} \mid \langle \text{select}, j \rangle \rightarrow \text{QUEUES}_{<} \right)
\]

\[
\text{QUEUES}_{<} = \left(\langle \text{enqueue}, x \rangle \rightarrow \text{if } \#s + 1 < NQ \text{ then } \text{QUEUES}_{<} \mid \langle \text{select}, j \rangle \mid \text{ QUEUES}_{<} \right)
\]

Note that these queues are bounded beneath \( NQ \) elements. Attempts to enqueue messages on filled queues result in dropped messages. The model combines many \( QUEUES \) subprocesses into a collection processes called \( QUEUESET \):

\[
\text{QUEUES}_{<} = \{i : \text{QUEUES} \mid i \in P \}
\]

The process called \( QUEUES \) communicates with kernel processes. Recall that \( i.q \) is the channel shared between \( i : K \) and \( i : QUEUES\):

\[
\text{QUEUES} = s : \text{QUEUESET} \parallel \text{ sel } : \text{ QUEUES } \parallel \mu X \bullet (q? (\text{enqueue}, j, m) \rightarrow s.j!(\text{enqueue}, m) \rightarrow X \mid q?(\text{dequeue}, j) \rightarrow s.j!(\text{dequeue}, j)?m \rightarrow q!m \rightarrow X \mid q?(\text{select}, Y) \rightarrow sel!(Z) \rightarrow q!Z \rightarrow X \mid q?(\text{clear}) \rightarrow \text{ QUEUES} )
\]

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.

$$QSELECT_s = Z : \text{VAR} \not\in ?Y \rightarrow$$

$$Z := \{\};$$

$$\mu X \cdot (\text{if } Y = \{\})$$

then

$$IZ \rightarrow QSELECT_s$$

else

$$j \in Y ;$$

$$Y := Y - \{j\} ;$$

$$(s,j) \mapsto (s,j ? A \rightarrow$$

$$ZZ = Z \cup A; X))$$

C. Determining Tag Sizes

We can solve for how big $\alpha$ must be in terms of $\beta$ and $\epsilon$. Partitioning requires that functions from $G$ must be injective, giving $2^{k^3} \leq 2^n$, or equivalently, $\alpha \geq 4/3$. As for unpredictability, $g$ is chosen randomly from $G$, so it will output elements in $\{0,1\}^n$ in random order. After $2^n$ calls, $g$ outputs elements from a set sized $2^n - 2^d$ at random. Since $\alpha \geq 4/3$, this “restricted” range for $g$ still has well in excess of $2^{n-1}$ elements. Failure occurs when a process can predict the output of $g$, which happens with probability no greater than $2^{n-1}$. Thus, $\alpha - 1 \geq \epsilon$. Combining these two restrictions, $\alpha \geq \max(\epsilon + 1, 4/3)$. For $\beta = 80$ and $\epsilon = 100$ we get $\alpha = 320$.

D. Proof

1) Alphabets: The relevant high, mid and low alphabets were defined in Section 4-A2. The rest of the events in the Flume model (like communication through the switch, to the process tag manager, etc.) are all hidden by the CSP-hiding operators, as given in Section IV-G. 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:

$$C_i \triangleq \{\text{i.s.}(S, I, \text{create_tag}, w) \mid$$

$$S, I \subset T \land w \in \{\text{Add}, \text{Remove}, \text{None}\}\} \cup$$

$$\{\text{i.s.}(S, I, \text{change_label}, w, L) \mid$$

$$S, I, L \subset T \land w \in \{\text{Integrity}, \text{Secrecy}\}\} \cup$$

$$\{\text{i.s.}(S, I, \text{get_label}, w) \mid$$

$$w \in \{\text{Integrity}, \text{Secrecy}\}\} \cup \ldots$$

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

$$R_i \triangleq \{\text{i.s.}(S, I, t) \mid$$

$$S, I \subset T \land t \in T\} \cup$$

$$\{\text{i.s.}(S, I, r) \mid$$

$$S, I \subset T \land r \in \{\text{Ok}, \text{Error}\}\} \cup$$

$$\{\text{i.s.}(S, I, L) \mid$$

$$S, I, L \subset T\} \cup$$

$$\{\text{i.s.}(S, I, O) \mid$$

$$S, I \subset T \land O \in O\} \cup$$

$$\{\text{i.s.}(S, I, p) \mid$$

$$S, I \subset T \land p \in P\}$$

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

$$L_i[P] \triangleq S_F[P] \mid (LO_i - MID_i)$$

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

$$\lambda(tr) \triangleq \#(tr \mid (LO_i \cup MID_i))$$

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 $\lambda(tr) = \lambda(tr') = k$.

a) Base Case: For the base case, consider all $tr, tr' \in \text{traces}(SYS)$ such that $\lambda(tr) = \lambda(tr') = 0$. In other words, $tr, tr' \in (HI_i - MID_i)^*$.

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 $\text{init}$, which is already waiting to accept incoming system calls when the system starts. By construction $S_{\text{init}} = \{\}$ and $L_{\text{init}} = T$. Since $t \notin S_{\text{init}}$, $C_{\text{init}} \cup R_{\text{init}} \subseteq LO_i$. Therefore, the system refuses all high events at startup, and $tr = \{\}$ is the only trace of SYS without low symbols (and for which $\lambda(tr) = 0$). For $tr = tr' = \{\}$, the claim trivially holds.

b) Inductive Step, $l \notin MID_i$: For the inductive step, assume the claim holds for all traces $tr, tr'$ of SYS such that $tr \cup (LO_i \cup MID_i) = tr' \cup (LO_i \cup MID_i)$ 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 $tr \in \text{traces(SYS)}$ such that $\lambda(tr) = k$, write $tr$ in the form $tr = p \land l \land h$, where $p$ is a prefix of $tr$, $l \in LO_i \cup MID_i$, is a single low event, and $h \in (HI_i - MID_i)^*$ are traces of high events. There are two cases: $l \notin MID_i$ and $l \in MID_i$. Consider the first here ($l \notin MID_i$), and see Section D2c below for the second.

Write the right trace in the same form: $tr' = p' \land l' \land h'$. It suffices to show that $L_i[S/tr] = L_i[S/(p \land l)]$. Indeed, if we have shown this equality for arbitrary $tr$, then the same applies for $S/tr'$, meaning $L_i[S/tr'] = L_i[S/(p' \land l')]$. By inductive hypothesis, $L_i[S/p] = L_i[S/p']$, and therefore $L_i[S/(p \land l)] = L_i[S/(p' \land l')]$.

By transitivity, we have that $L_i[S/tr] = L_i[S/tr']$, 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, …) where $l \notin S$. After this event, $i$-$K$ is in a state ready to receive a new system call (i : KS,1,0). Because all events in $h$ are high events, none are system calls of the form i.s.(S, I, O, …) with $l \notin S$, 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:

$$L_i[SYS/(p \land l \land h)] = L_i[SYS/(p \land h \land l)].$$

We can apply the inductive hypothesis to deduce that:

$$L_i[SYS/(p \land h \land l)] = L_i[SYS/p]$$

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

$$L_i[SYS/(p \land h \land l)] = L_i[SYS/(p \land l)]$$

and by transitivity:

$$L_i[SYS/(p \land h \land l)] = L_i[SYS/(p \land l)]$$

which proves the claim for this case.

Note two special events here: first, a return from create_tag in which $i$ receives the special tag $t$; and second, a return from
The extended definition of noninterference.

Thus, there are three state transitions that can alter the reply to the receive system call: i.s.(S, I, O, create_tag, w) for some i ∈ P, and some w ∈ {Add, Remove, 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 ∈ LO_i, it follows that t ∈ 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 SF[S/tr]) if the space of t’s returned somehow depends on h, because h changed the state of the shared tag manager.

We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The only way an event in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The event after in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The only way an event in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The event after in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The only way an event in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The event after in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The only way an event in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.

The only way an event in h can influence the outcome r is to alter the composition of O, which the tag manager checks on i’s behalf by answering i.q.(check+) and i.q.(check-) within the CHECK subprocess. That is, some high process j must request a new tag with system call e = j.g(S', I', O', 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 ∈ h is a high event, t ∈ S'. Because l is a low event, t ∉ S. Thus, S' ≠ S, and assuming g is injective, it follows that t' ≠ t, for all x. Therefore, events in h cannot influence which tags t’ might be allocated as a result of a call to create_tag. We apply the same argument as above, that h and l can happen in the system either before or after one another without changing the failures of the system.
We have covered all of the relevant cases, and the theorem follows by induction.

E. *select* and Timing

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

• \( Y \leftarrow \text{select}(t, X) \)

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

A new kernel subprocess \( \text{SELECT} \) handles the new system call; it allows a user program to wait for the first available receive channel to become readable:

\[
\begin{align*}
\text{SELECT}_{S,I,O} &= (s?\langle \text{select}, t, A \rangle \rightarrow \\
&\hspace{1cm} (\mu X \cdot (q!(\text{select}, A) \rightarrow q?B \rightarrow \\
&\hspace{2cm} \text{if } B = \{} \hspace{1cm} \text{then } INTRECV^*_{S,I,O} \cdot X \\
&\hspace{2cm} \text{else } s^1_B \rightarrow K_{S,I,O}) \\
&\hspace{1cm} \triangle_t (s^1_\{} \rightarrow K_{S,I,O}))
\end{align*}
\]

There are three new CSP operators here. The first is \( \mu X \cdot F(X) \), 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 \( \text{SKIP} \). 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 \( \text{SELECT} \) calls subprocess \( INTRECV^* \), which behaves mostly like \( INTRECV \), except it keeps receiving until an admissible message arrives:

\[
\begin{align*}
INTRECV^*_{S,I,O} &= c?(S_a, I_a, j, X, m) \rightarrow \\
&\hspace{1cm} g!(\text{dual_prives}, O) \rightarrow g?D \rightarrow \\
&\hspace{2cm} \text{if } (S_a \subseteq S \cup D) \land (I - D \subseteq I_a) \\
&\hspace{2cm} \text{then } q!(\text{enqueue}, (X, m)) \rightarrow \text{SKIP} \\
&\hspace{2cm} \text{else } INTRECV^*_{S,I,O}
\end{align*}
\]

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 \( \text{tick} \), which represents a discrete unit of time’s passage. In particular, Schneider provides the \( \Psi \) function for mapping processes from timed CSP to discrete-event CSP with a \( \text{tick} \) event representing the passage of time. In the proof of noninterference, consider \( \text{tick} \) a low event, that is not hidden by any concealment operator. Then apply the \( \Psi \) translation to all states of the Flume model.