Decentralized Information Flow Control for Operating Systems – A Survey

Kenneth Ezirim, Wai Khoo, George Koumantaris, Raymond Law, and Irippuge Milinda Perera

The Graduate Center of CUNY
{kezirim,wkhoo,gkoumantaris,rlaw,iperera}@gc.cuny.edu

December 10, 2012

Abstract. Decentralized information flow control (DIFC) is a model that guarantees absolute end-to-end security. With DIFC Operating systems can have total control of system resources. On top of that, Operating Systems can monitoring the flow of information through a set of rules and policies. This paper describes two Decentralized information flow control (DIFC) operating systems, Flume and HiStar. Flume eases the use of DFIC in existing applications and allows safe interaction between conventional and DIFC-aware processes. HiStar is more strict in terms of information flow control and allows users to use data security policies without affecting or changing the structure of applications. This paper will further discuss whether HiStar or the Flume model can be implemented as new operating systems. Improvements to the current security model will be suggested taking into consideration the amended security additions that HiStar and Flume have to offer.

1 Introduction

1.1 Decentralized Information Flow Control (DIFC)

Decentralized Information Flow Control (DIFC) for Operating Systems is a model for controlling information flow in systems with mutual distrust and decentralized authority. It is an approach to security that allows software application writers to control how data flows between the pieces of an application and the outside world. Untrustworthy software are allowed to compute but trusted security code controls the release of that data [1–3].

As an example of the Decentralized Information Flow Control (DIFC) model, consider Alice and Bob who want to schedule a meeting while keeping their calendars mostly secret [1]. Alice and Bob each place a secrecy label on their calendar file, and then only a thread with those secrecy labels can read it. When the thread is ready to output an acceptable meeting time, it must call a function that then declassifies the result [1]. The declassification function checks that its output contains no secret information. For example, the output is simply a date and does not include Bobs upcoming visit to the doctor [1].

In the Alice and Bob example, he secrecy labels ensure that any program that can read the data cannot leak the data, whether accidentally or intentionally [2, 3]. The secrecy label is tied to the data, and it restricts who may access the data. The decision to declassify is localized to a small piece of code that can be closely audited [1, 3].

1.2 Existing Models of Operating System Security

The existing models for OS security are inadequate. Regular users are moving away from the traditional computer environment. There is a shift from a desktop related computer environment

to a web-server/cloud related environment. Users store their data in the cloud using services like VMware, Citrix, NetApp, EMC [3, 4]. The need for applications to run on the web-servers is the same, therefore we need to find a method to secure our data [4, 5]. Operating system security abstractions, such as file permissions and user IDs, are too coarse to express many desirable policies, such as protecting a user's financial data from a mistrusted browser plug-in. For example, if a user B is allowed to read A's data, A cannot controls how B distributes the information it has read, therefore A looses integrity/control of the data. Control of information propagation is supported by HiStar & Flume, surveyed on this paper [5].

The current security setup on the cloud supersedes the old one(Regular home Desktop user setup) and does not take into consideration the needs of the cloud [5]. The average number of serious vulnerabilities introduced to websites by developers in 2011 was 148, down from 230 in 2010 and 480 in 2009. Vulnerabilities could be found by a hacker, resulting in a high-profile data breach such as those that affected Sony, Symantec, and AT&T.

1.3 HiStar & Flume

This paper presents HiStar and Flume, an OS-level protection to provide Decentralized Information Flow Control (DIFC). At a high level, these two OS's control which messages sent to a machine can affect which messages sent from the machine, thereby letting us put together secure systems out of untrustworthy components [6]. HiStar has the luxury of an entirely trusted kernel with thread, address space, segments, gates, containers, and devices labels [3, 6, 7]. Flume takes it a step further by ensuring the need to specify how and when they use their privileges to label flows and also makes sure that no process can have both an uncontrolled channel and access to private data it cannot declassify [3, 7].

2 HiStar Architecture

Like most operating systems, HiStar adopts the same operating system abstraction based on six-level kernel object types namely: threads, address spaces, segments, gates, containers and devices. HiStar treat each of these objects as a separate entity, assigning labels and clearances that control the flow of information between the objects. Assigned label either specifies whether an object has an untainting privileges for each category of taint or how tainted an object is in that category.

Every operation in HiStar requires the kernel to check whether information can flow from one kernel object to another. Information flow from an object with label L_1 to another labeled L_2 is allowed if and only if label L_1 is less tainted than label L_2.

$$L_1 \sqsubseteq L_2 \text{ iff } \forall c : L_1(c) \leq L_2(c)$$

In order to grant some objects untainting privileges the following symbols are used describe how the ownership level of a category should be treated in both cases of writing and reading another object.

- $\star$ to lower the ownership level of a category such that a object with untainting rights can write to another object.
- ✪ - to raise the ownership level of a category so that an object can observe the tainted information of another object.

The introduction of the symbol ✪ creates six levels of ownership, ordered in the following way: $\star < 0 < 1 < 2 < 3 <$ ✪. ✪ is actually used in access rules and does not appear in labels of actual objects. The label $L = L_1 \sqcup L_2$ is interpreted in terms of category as follows: the label on category c given that $L(c) = max(L_1(c), L_2(c))$. In other words, the relationship $L_1 \sqcup L_2$ is described as the least upper bound of the two labels L_1 and L_2.

Every object in HiStar is characterized by a unique, 61-bit object *ID*, a *label*, a *quota* bounding its storage usage, 64-byte of mutable, user-defined *metadata* (used, for instance, to track modification time) and a few *flags* such as an immutable flag that makes an object irrevocably read-only. With the exception of threads, an object's label is specified upon creation and becomes immutable afterwards. Some object might allow copies with different labels, which is useful especially in the case of re-labeling. The simplest kernel object is a segment which is similar to a the concept of file in other operating systems. A segment represents a variable length byte array used mainly for data storage.

In HiStar, an object's label controls the flow of information to and from another object. The interface is designed such that the following property is always maintained:

> *The contents of object A can only affect object B if, for every category c in which A is more tainted than B, a thread owing c takes part in the process.*

This implies that a system component can affect others without perfect understanding of either the components or their interaction with the system. Applications are structured such that key categories are owned by small amount of critical code, removing the huge responsibility of having to monitor the entire code. The critical codes are responsible for pre-authorizing actions on an application's behalf by modifying taints for an object in the category of interest.

To prevent code from unsanctioned access or replication of private data, threads are accorded a clearance label. A clearance label specifies the upper bound on the thread's label and the labels of objects that the thread would like allocate or grant access to. For example an object with clearance of $\{2\}$ cannot read from an object tainted $\{3\}$ in the read category because its clearance prevents it from tainting itself $\{3\}$ in order to perform the read operation.

HiStar has a single-level store - the entire system is restored from the most recent on-disk snapshot. This eliminates the need to re-initialize processes such as daemons on booting the operating system. This mechanism allows for efficiency to implement the file system with the same kernel abstraction as the virtual memory. The disadvantage is that one can no longer rely on re-booting to kill off errant application and reclaim system resources. The single-level store in HiStar actually enhances disk usage as kernel objects are written to disk at each snapshot and can be deallocated from memory once stably stored. With the enforcement of object quotas, the problem of disk space exhaustion is further minimized, as quotas form a hierarchy under the control of the system administrator.

2.1 HiStar Kernel Components

In this section, we discuss the components of the HiStar kernel. The section is divided into subsections that discuss in details threads, containers, quotas, address spaces and gates as they apply to the HiStar architecture. The security on the flow of information between kernel objects are enforced using labels with clearance label restricting the level of taint an object can attain.

Threads A thread is a kernel object characterized by a label L_T and a clearance label C_T. A thread T has $L_T(c) = \mathbf{1}$ and $C_T = \mathbf{2}$ for a category c. The system call to create a thread is

- cat_t *create_category*(void)

The system call pseudo-randomly chooses a previously used category c and sets $L_T(c) \leftarrow \star$ and $C_T(c) \leftarrow 3$. At this point T is the only thread whose label maps c to a value below the system default $\mathbf{1}$. Therefore, no thread has any inherent privileges with respect to categories created by other threads. Nevertheless, T can raise its own label through the system call

- int *self_set_label*(label_t L)

which sets $L_T \leftarrow L$ provided that the following condition is satisfied $L_T \sqsubseteq L \sqsubseteq C_T$

This modification can enable T to read a tainted object. T can as well increase its clearance using the following system call

- int *self_set_clearance*(label_t C)

This system call is successful only if $L_T \sqsubseteq C \sqsubseteq (C_T \sqcup L_T^{\circledast})$

HiStar imposes some restrictions on threads with respect to their labels and the objects that they want to access. The restrictions are

- T can observe object O only if $L_O \sqsubseteq L_T^{\circledast}$ (i.e. no read up)
- T can modify object O, which in HiStar implies observing O, only if $L_T \sqsubseteq L_O \sqsubseteq L_T^{\circledast}$ (i.e. no write down).

$L_T \sqsubseteq L_O$ implies that the thread T can write to the object to modify it and $L_O \sqsubseteq L_T^{\circledast}$ allows the thread T to also read from the object. These restriction is repeatedly applied in HiStar's specifications and abstractions.

Containers A container is an abstraction in HiStar to enable hierarchical control over object allocation and deallocation. Similar to Unix directories, containers hold hard links to objects. The root container is a specially designated container that can never be deallocated. Objects are deallocated from a container once there is no existing path to that object from the root container. Possible links between containers and other objects type is presented in the Fig. 1.

When a thread allocates an object, it must specify the container for the object as well as a 32-byte descriptive string stating the object's purpose. The system call to create a container is

- id_t *container_create*(id_t D, label_t L, char* *descrip*, int *avoid_types*, uint64_t *quota*).

D is the ID of an existing container in which a new container will be created, L is the desired label for the newly created container, *descrip* is the descriptive string. *avoid_types* is a bitmask specifying the types of kernel objects that cannot be created in the container or in any of its descendants. Quota *quota* is addressed in a separate section.

Directories are implemented in HiStar using containers. The containers can be traversed, started from the root container, in a similar way as in other file systems using container ID. A separate segment in each directory container is used to store file names.

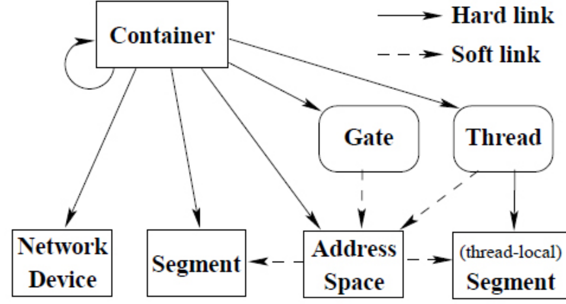

Fig. 1. Kernel object types in HiStar. Soft links name objects by a particular ⟨container ID, object ID⟩ container entry. Threads and gates, which can own categories (i.e., contain * in their labels), are shown by rounded rectangles.

A thread T can create a hard link to a segment S in container D provided it can write to D ($L_T \sqsubseteq L_D \sqsubseteq L_T^{\circledast}$) and its clearance is high enough to allocate objects at S's label ($L_S \sqsubseteq C_T$). Thus T can prolong S's life even without permission to modify S. Most system calls specify objects not by ID but by ⟨container ID, object ID⟩ pairs, referred to as container entries. For any thread T to use container entry $\langle D, S \rangle$, D must contain a link to S and T must be able to read D i.e. $L_D \sqsubseteq L_T$. For another thread T' to have access to actions by T, the following condition must be satisfied $L_T \sqsubseteq L_{T'}$. Container entries allows the kernel to control the ability of a thread to know about an object's existence. A thread must possess the required permission to read the labels object. After examining the label of an object more tainted than itself, a thread can decide how to taint itself in order to read that object.

In HiStar, there is a special notion that every container contains itself. Therefore a thread T can access a container as $\langle D, D \rangle$ when $L_D \sqsubseteq L_T^{\circledast}$ even if it cannot read the D's parent container D'. The allocation rule for the creation of such a container D in D' is that the thread T' that created D must own every category c for which $L_D(c) < L_{D'}(c)$ i.e. $L_{T'} \sqsubseteq L_{D'} \sqsubseteq L_{T'}^{\circledast}$.

Quotas Every object in HiStar kernel has a quota, which states the limits of its storage space. The root container is reserved a quota of reserved value inf. A container's quota, therefore, is a cumulative of all data structures therein and quotas of objects contained in it. Quotas are specified during object creation and can be modified using the system call

- int *quota_move*(id_t D, id_t O, int64_t n)

which adds n bytes to both O's quota and D's usage and D must contain O. The thread T initiating this system call must satisfy the following condition $L_T \sqsubseteq L_D \sqsubseteq L_T^{\circledast}$ and $L_T \sqsubseteq L_O \sqsubseteq C_T$ for it to be successful. In order to convey information back to T in the case where there is not enough byte to allocated, $L_O \sqsubseteq L_T^{\circledast}$ must hold.

HiStar supports the existence of hard links to threads and segments in multiple containers but their quota is added to each container's usage. But links are not allowed on object whose quotas may subsequently change. The kernel enforces this by using "fixed-quota" flags on each object. The flag must be set before adding a link to an object and can never be cleared.

Address Spaces Address spaces are objects that associated with threads containing a list of page-aligned virtual addresses (VA). VA is represented by the following mappings

$$VA \rightarrow \langle S, offset, npages, flags \rangle$$

S is a container entry, $offset$ and $npages$ specify a subset of S to be mapped, $flags$ specify read, write and execute permission.

Like every other object, an address space is assigned a label L_A to which label rules apply. Threads can modify an address space A only if

$$L_T \sqsubseteq L_A \sqsubseteq L_T^{✪}$$

and can observe or use A if $L_A \sqsubseteq L_T^{✪}$. To lunch a new thread, one is required to specify its address space and entry point. Threads can switch address via the usage of the system call $self_set_as$.

Every thread has a one-page local segment than can be mapped in its address space using a reserved object ID. The local segment served as reserved space when other part of the virtual space are not accessible. It can be used by a thread raising its label as a temporary stack while making copies of its address space.

In HiStar, threads can send alert to each other, which raises an exception and initiate the alert handler. This operation is only possible if a thread T_1 can write to the address space A of a thread T_2 i.e. $L_{T_1} \sqsubseteq L_A \sqsubseteq L_{T_1}^{✪}$ and also to observe T_2 i.e. $L_{T_2} \sqsubseteq L_{T_1}^{✪}$. These condition suffices to enable T_1 to gain full control of T_2 as well enable T_2 to send information to T_1.

Gates Gates are kernel objects that provide protected control transfer among threads. Gates makes it possible for threads to jump to a pre-defined entry point in an entirely different address space with the help of an additional privilege. A gate object has the following:

- a label L_G, which may contain $\star$ just like threads,
- a clearance label C_G
- a thread state, including the container entry of an address space
- an initial entry point
- initial stack pointer
- some closure arguments to pas the entry point function.

A thread T can allocate a gate G whose label and clearance satisfy the condition

$$L_T \sqsubseteq L_G \sqsubseteq C_G \sqsubseteq C_T$$

A thread T invoking G must satisfy the following conditions

$$L_T \sqsubseteq C_G$$

$$L_T \sqsubseteq L_V$$

$$(L_T^{✪} \sqcup L_G^{✪}) \sqsubseteq L_R \sqsubseteq C_R \sqsubseteq (C_T \sqcup C_G)$$

where L_R and C_R are the requested label and clearance specified by T. L_V is a verify label provided by T to verify its ownership of categories. The entry point function examines the label L_V for additional access control. Gates play important role in HiStar. Gates can be used to transfer privileges especially during the login process. This aspect would be discussed further in user-level design.

3 HiStar Implementation

3.1 Hardware

HiStar requires a x86-64 bit processor to run. The required 64-bit processor allows the Operating System to utilize a vast amounts of virtual memory addressable spaces to be used for file descriptors. A B+ tree is used to map Object ID values in the labels to disk addressable space. [3] In essence the object ID works as an index to the various object primatives stored on disk. Because all meta data like object ID's, disk offsets, and flags have fixed size values, HiStar's allocation of memory is simplified.

3.2 Kernel

One of the main advantage of HiStar is its simple design paradime and small kernel. The fully trusted kernel has 15,200 lines of C code and 150 lines of assembly. [3] The code itself can be broken down into 4 main components, architecture specific code for virtual memory and threads, B+ tree implemention, device driver and DMA-based IDE support, and System Calls.

HiStar maintains two additional B+ trees in addition to the B+ tree used to map object ID's to virtual memory addresses. The two B+ trees are represent the remaining free space. The main purpose of these two trees is for memory management. Specifcally allocation of new memory, deallocation, and joining adjacent free blocks. Due to HiStar's technical design certain optimizations can be assumed, for instance any two immutable labels can have simple function results cached.

HiStars current design does put device drivers inside the fully-trusted kernal. This may pose a security issue, although if the device driver itself is crafted with the same idea of information flow, the only drawback is the increased code complexity. In future releases the designers have made plans to move the drivers outside the kernel.

3.3 User-Level Design

HiStars user level environment was based on Unix's design. HiStar implements Unix like features through a port of the uClibc library. [3] uClibc supplies a framework for most Linux type commands, which rests on a later above HiStar code. A user can fully utilize all linux commands and system calls as if they were running Linux. HiStar's primitve objects are abstracted into unix like file descriptors, processes, file systems, and fork and exec commands. These commands are all run on the user level, on top of HiStar so no security is surrendered.

4 Flume Architecture

Flume provides DIFS at the granularity of processes and integrates DIFC control on OS abstractions such as pipes, sockets and file descriptors.

Flume is built in user-space with small kernel patches for implementation convenience and portability. Flume implementation runs on Linux and BSD. This implies that Flume leverages the existing kernel components in these system to provide DIFS.

In flume, processes are divided into untrusted processes and trusted processes. Untrusted process do most of the computation. They are constained but may not be aware of the DIFS controls. Trusted processes are aware of the DIFS controls and constrain the untrusted processes by setting

up privacy and integrity controls. Trusted process have the privileges to violate the information flow pattern by declassifying private data and endorsing data as high integrity.

Just in HiStar, Flume leverages labels and tags to track data flow in the operating system. Tags are a set of opaque tokens that carries no inherent meaning. Processes generally associate tags with some category of secrecy or integrity. Labels on the other hand are subsets of tags. Labels form a lattice under the partial order of the subset relation. Each Flume process p has two labels - S_p for secrecy and I_p for integrity. If tag $t \in S_p$, then the system concludes that p has seen some private data and if tag $t \in I_p$, then every input to the process p is considered as having high integrity. Files, just like processes, can also have secrecy and integrity labels. A tag can appear in any type of label but cannot be in both at the same time because the secrecy and integrity usage pattern are so different.

4.1 Decentralized Privilege

Unlike in IFS where only a trusted entity can create new tags, subtract tags from secrecy labels and add tags to integrity labels, in Flume DIFS any process can create tags, which gives the process the privilege to declassify or endorse the created tags.

Flumes allows two capabilities per tag. A process can have its own set of capabilities O_p that enables to add a tag to a label or remove it from a label. For a tag there can be sets of capabilities: t^+ to add a tag to a label and t^- to remove a tag from a label.

Allocation of an arbitrary tag t yields a new set of capabilities granting p dual privilege for t

$$O_p \leftarrow O_p \cup \{t^+, t^-\}$$

Flume supports global capability set $\mathbf{O}$ which is a common capability set to all process in the system i.e. $\mathbf{O} \subseteq O_p$. A process can test whether a given capability is in $\mathbf{O}$ but to prevent data leaks, process prevented from listing the contents of $\mathbf{O}$. However a process can still enumerate its non-global capabilities i.e. $O_p - \mathbf{O}$.

Two processes are allowed to transfer capabilities provided they can communicate. A process can freely drop non-global capabilities but there are some restrictions to this. For a set of tags T, the capability set $\{T\}^+ = \{t^+ | t \in T\}$ and $\{T\}^- = \{t^- | t \in T\}$.

Export protection is a mechanism employed during the allocation of a new tag b. During the process of allocation the new tag b^+ is added to $\mathbf{O}$ where as only the trusted process gets b^-. This implies that any process can add b^+ to S_p therefore read b-secret data but only process that own b^- can declassify it and export it out of the system.

A related but much stringent policy is the read protection where neither t^+ nor t^- of the allocated secrecy tag is added to $\mathbf{O}$. By this means an allocating process can control the processes that has access to read t-secret data and those that can declassify the data.

In terms of integrity, after the allocating the integrity tag v, the allocating process, known as the certifier, adds v^- to $\mathbf{O}$ but keeps $v+$ to itself. The certifier alone can endorse information as high-integrity, other processes can only remove v from I_p.

4.2 Security

One of the assumptions of Flume is that all processes are running in the same machine and exchanging messages via flows. The main goal is to track data flow by regulating both communication between processes and changes in labels.

In Flume, safety is defined in terms of security and integrity label changes. A system is considered secured in Flume if and only if all allowed process label changes are safe and all allowed messages are safe. When a process requests a change, only those label changes permitted by a process's capabilities are safe. Given a process p, let L be either S_p or I_p and let L' be the new value of the label. The change from L to L' is safe if and only if

$$\{L' - L\}^{+} \cup \{L - L'\}^{-} \subseteq O_p$$

An example would be say a process p wishes to subtract a 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 $t^{-} \in O_p$, i.e. p has a subtraction capability.

Safety of messages exchanges is handled in the same manner as in HiStar, where flows are disallowed from more tainted to less tainted objects("no read up" and "no write down" constraints). In classical IFC, p can send a message to another process q if only $S_p \subseteq S_q$ and $I_q \subseteq I_p$. Processes might be restricted to perform read or write but if they can change their labels and relax their rules, they might be able carry out these operations. This would involve process changing their labels, using centralized rules to send messages and then restoring their labels once the message is received, without any permanent label changes. Flumes allows temporary label changes only for tags in the dual privilege label D_p. Therefore a message from p to q would be safe if and only if

$$S_p - D_p \subseteq S_q \cup D_q \text{ and } I_q - D_q \subseteq I_p \cup D_p$$

$S_p - D_p$ implies p reducing its secrecy level thus declassifying its message, and $I_p \cup D_p$ implies p raising its integrity level for q thus endorsing the message it is sending. q increases its secrecy level to $S_q \cup D_q$ in order to be able to receive data from p and endorses the received with integrity $I_q - D_q$.

Any external process x outside of Flume's control, such as remote host, user's terminal or a printer, has an empty security and integrity label $S_x = I_x = \{\}$ and $O_x = \mathbf{O}$. Therefore, processes can only write to the network only if they can reduce their secrecy label $\{\}$ because $S_p \subseteq S_x$ and can read from the network or keyboard if only $I_x \subseteq I_p$.

Generally, objects such as files and directories are modeled in Flume as a process. This means that objects like any other process are characterized by their security and integrity label. A process p attempt to write to an object o becomes a flow from p to o which is reading a flow by o from p. The operation is subject to restrictions that permits the process p to write to o. In some cases, such as the creation of a file in a particular, process p must abide by the rules of accessing the directories which grants it access to eventually create a file.

4.3 Endpoints in Flume

In the section, the general guidelines for the Flume model design is described, providing criteria for the system implementing the model to be considered secured. The goal of the Flume system is to fit the existing APIs for process communication while upholding security in the Flume model.

The Flume system applies restriction to file descriptors used as primitives for communication in Unix systems. The descriptor is assigned an endpoint. An endpoint is characterized by label settings that control the reading and writing. A process capable of adjusting the label setting can control the information via the file descriptor. The introduction of endpoints helps to simplify application development. Failure to deliver a message are failed silently such that the process involved can

simply log the error, thereby enabling the programmer to debug the application. Endpoints help to make many declassification or endorsement decision explicit. File descriptors that serve as avenues for declassification or endorsement must be explicitly marked by the processes that are using them.

Endpoints, like processes, have security and integrity labels. A process in a quest to acquire a new file descriptor must provide a new endpoint. An endpoint e by default has $S_e = S_p$ and $I_e = I_p$. A process can have readable and writable endpoints for the resources that it owns. A readable endpoint e is considered safe if and only if $(S_e - S_p) \cup (I_p - I_e) \subseteq D_p$ and a writable endpoint is considered safe if and only if $(S_p - S_e) \cup (I_e - I_p) \subseteq D_p$. A read/write endpoint is considered safe only if both conditions are met. Therefore, message from an endpoint e to another f is considered safe only if e is writable and f is readable, $S_e \subseteq S_f$ and $I_f \subseteq I_e$

Endpoints are very effective in control communication between processes over sockets and pipes. They are used to ensure that there no leakage of information by silently dropping data if it is unsafe. The receiving process has no way of distinguishing between an unsent or dropped message because it is unsafe, thereby protecting the message from eavesdropping. Even an attempt by a process to use a mutable endpoint in an unsafe way is caught by the system and informs the process of the failure and its specific cause. Bi-directional data flow is also supported in using endpoints.

Endpoints are also applied in a process's file I/O with coarse granularity. When a process opens a file f it specifies the labels that apply to the endpoint e_f. If no labels are specified for e_f, they default to p's. Reading file f is only successful if the e_f is a safe readable endpoint i.e. $S_f \subseteq S_{e_f}$ and $I_{e_f} \subseteq I_f$. For successful writing to file f by p, e_f must be a safe writable endpoint $S_{e_f} \subseteq S_f$ and $I_f \subseteq I_{e_f}$

Immutable endpoints allow Flume to send and receive data in the system via network connections, user terminal and the like. The system assigns immutable read/write endpoint to a process if discovers that the process has access to resources that allow access to transmission or reception of messages. By default an endpoint e is assigned security and integrity equivalent to $S_e = I_e = \{\}$. Since e must always be safe, then the following conditions must be satisfied $S_p - D_p = I_p - D_p = \{\}$ for the process to have the privilege to import or export data via network.

5 Flume Implementation

Flume's design is a component within the kernel rather than an entire Operating System like HiStar. It is immediately visible that, any process running outside of Flume is vulnerable because of this. Currently there are two different implementations of Flume, one for Linux and one for OpenBSD. The Linux implementation runs as a component within the kernel, while the OpenBSD version utilizes systrace system calls.

5.1 Kernel

The Linux Security Module (LSM) is composed of a Reference Monitor, a dedicated spawner, a tag registry, and user space file servers, all of which work together to manage labels in Flume. [7] The Reference Monitor acts as a gateway, between the kernel and processes, as it is the only component that has direct communication with outside processes. Label tracking and authorization is done through the Reference Monitor because of this.

To perform any computing on flume a processmust communicate with the reference monitor through remote procedure calls sent over a control socket. Flume does provide a basic API for

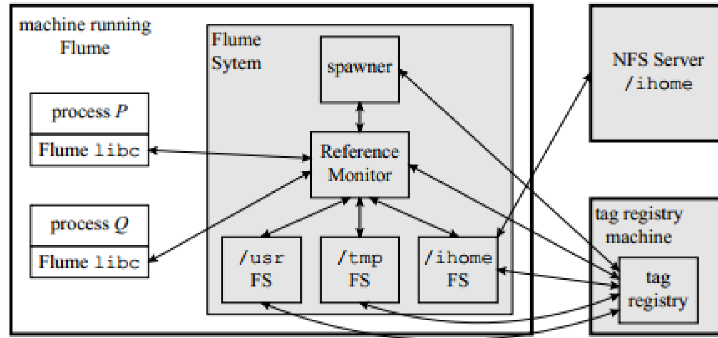

Fig. 2. The main components of Flume architecture. The shaded components reflect Flume's trusted region.

standard file descriptor methods like open. Multithreading is accomplished by allowing multiple control sockets.

5.2 Confined and Unconfied Processes

Each process is either confined and running within Flume, or unconfined and running outside of Flume. Unconfied processes are considered outside Flume's reach and can be accessed with standard Linux methods. This is fine for less sensitive data, but sensative computing should be completed as a confined process. A confined process must go through the Reference Monitor in order to use any system calls and resources. Depending on label values present, processes fit three basic categories, either the process is allowed to run normally, it is not allowed but Flume runs the commands on it's behalf, or the process is blocked entirely.

Any confined process is run as an unprivlidged user in Linux. In essence even if an attacker managed to take over a confined process, they could only use the same methods dictated by Flume's LSM policy.

Flume's LSM policy disallows all direct access to file systems by confined processes. Fork is blocked in Flume, due to the fact that all process spawning must occur from within the dedicated spawner. Instead the creators offer two commands flume_pipe and flume_socketpair which performs a safe version that communicates through the Reference Monitor to call the spawn device. These commands return back a file descriptor that can be used normally.

6 Applications

In this section we survey some of the interesting applications allowed by the two DIFC mechanisms. It should be noted that these applications are not unique to the DIFC systems, rather, they are much easier to be constructed within a DIFC environment. The high level idea of these applications is to minimize the trusted code to an easily manageable size with the help of the underlying DIFC system. First, we survey three applications of HiStar. Followed by that is an application of Flume.

6.1 Applications of HiStar

Anti-Virus Software A typical anti-virus software requires a high level of privileges than the regular software in order to perform the scanning/quarantining of the infected files successfully.

The problem with this approach is that if the anti-virus software itself is malicious, the user has no guarantee of security. For example, the software can send the user's personal data to a third party server through a covert channel.

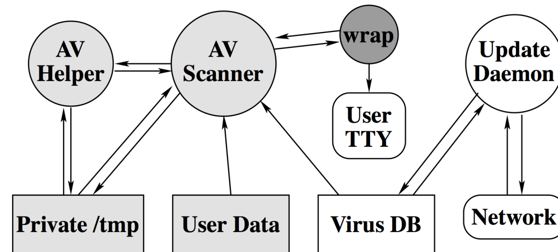

Fig. 3. The *untrusted* anti-virus software constructed with HiStar. HiStar prevents information flowing from lightly-shaded components to the unshaded. The strongly-shaded component has privileges to relay output to the terminal.

The first application of HiStar is an *untrusted* virus scanner whose access to user's files is tightly controlled by the HiStar's DIFC service. As you can see from Fig. 3, HiStar prevents the scanner from flowing the information about the user's data to the outside of the system by *tainting* them as sensitive. Although the scanner has reading and quarantining abilities to the user's files, it's not allowed to transport the files to the external entities.

The way the authors have implemented this anti-virus software is by porting the freely available ClamAV [8] open-source anti-virus software. The authors claim that they were able to implement the untrusted version of the software with very minimal change to the existing code. Specifically, they could move only the access control code to the HiStar service while keeping the features such as updating the virus database intact. This seems to be a good sign for porting other software to untrusted versions in the future.

User Authentication Most operating systems require a highly-trusted process with superuser permissions to manage user authentication. As shown in the previous section, it is desirable to make that process less trusted in order to avoid unnecessary breaches of user privacy. The new user authentication application implemented by the authors of HiStar does exactly that. This new system makes sure that even if the user accidentally provides his password to a malicious authentication service, only one bit of information about the user's password is reveled.

As you can see from Fig. 4, this new authentication system consists of four components, none of which requires superuser privileges. The *login* component initiates the authentication by asking the *directory service* for a handle to the user's authentication daemon. Next, the *login* service contacts the *user authentication service* in order to verify the user's password. Both the *directory service* and the *user authentication service* have access to a *logging service* to record all the authentications that take place.

VPN Isolation Nowadays, it is common for people to connect their computers to otherwise firewalled networks through encrypted virtual private networks (VPN). The use of encryption, however, does not prevent the risk of having some malware either infect internal machines or

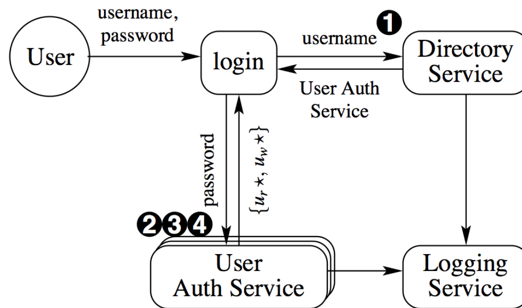

Fig. 4. A high-level overview of the user authentication system realized through HiStar. None of the components require superuser privileges.

divulge sensitive files to the world. Therefore, it is desirable to have a VPN service that provides the guarantees beyond what is provided by the point-to-point encryption.

The third application of HiStar is a VPN isolation service that attempts to mitigate the problems stated above. The authors have implemented this VPN service by porting the freely available OpenVPN software. The general idea of this implementation is as follows. The VPN client component labels all the incoming data through the VPN connection with a label whose sensitivity level is higher than that of the network interfaces. Since HiStar prevents information flowing from high-sensitive components to low-sensitive ones, the malware cannot reveal the user's data coming from the VPN connection to the outside world.

6.2 An Application of Flume: MoinMoin Wiki

The application of Flume we are surveying is a Python-based Web publishing system (*i.e.*, "wiki") that allows the clients to read and edit the pages hosted at a web server. The original application (MoinMoin) used very basic access control lists (ACLs) to govern which users and groups can modify the individual pages of the wiki. The problem was that MoinMoin comprises over 91,000 lines of code in 349 modules. It checked ACLs in 41 places across 22 modules. It goes without saying that it is highly probable that some security vulnerabilities could be left out while auditing MoinMoin's lengthy source code.

The authors of Flume have ported MoinMoin to include Flume for providing access control of the users. They were able to do this by change only 2% of the original MoinMoin source code. The ported version is about 30%-40% slower than the original version due to overheads of the Flume implementation. But, they were able to confine the security related code to a very small code base and also incorporate additional security policies to the ACLs.

7 Performance

With a new operating system, especially one with added complexity such as security, evaluating its performance with current state of the art is imperative. Both HiStar [3] and Flume [7] compared their systems with some Linux version. HiStar compared it with Linux and OpenBSD under several benchmarks [7]. While Flume compared it with Linux version 2.6.17 with Apache server attached [3]. Let's first discuss about HiStar in Section 7.1, follow by Flume in Section 7.2.

7.1 HiStar

HiStar runs their benchmark tests on three identical system, each with a 2.4 GHz processor, 1 GB memory, and 7200 RPM hard drive. The first machine ran HiStar, the second ran Fedora, and the third ran 32-bit OpenBSD with an in-memory mfs file system. Their benchmark tests include

1. Inter-Process Communication (IPC), which measures the latency of communication over a Unix pipe
2. Fork/exec, which measures the latency of executing */bin/true*.
3. Log-structured File System (LFS) small-file
4. LFS large-file

For IPC benchmark, two processes are created and connected by two uni-directional pipes. That is each process send messages it receives back to the other process. The algorithm then sends 8-byte messages back and forth, making over one million round-trips, then it measures the average round-trip time. For this benchmark, HiStar performs better than Linux, but somewhat slower than OpenBSD [3].

For fork/exec benchmark, HiStar did not perform as well as others, partly because Linux and OpenBSD pre-zero memory pages, which HiStar does not. In OpenBSD and Linux, it takes 9 system calls to fork a child, that includes executing */bin/true*, having */bin/true* exit, and having the parent wait for the child. On the other hand, HiStar requires 317 system calls on top of HiStar's low-level interface. The advantage of its low-level interface is that it provides flexibility to implement more efficient library calls such as *spawn*. *Spawn* starts a new process running a specified executable. This function runs about 3 times faster than fork/exec [3].

For LFS small-file benchmark, it operates on 10000 1kB-sized files and compute the total running time for creates, reads, and unlinks. HiStar has comparable performance for the asynchronous and cached variations. Linux, however, outperforms HiStar in uncached read, averaging less than $\frac{1}{10}$ the disk's 8.3 msec rotational latency to read each file. The authors believe that this performance is due to read lookahead in the IDE disk because Linux group files from the same directory while HiStar does not. Disabling this feature, HiStar and Linux perform comparably [3]. In synchronous unlink phase, HiStar performs worse than Linux because HiStar implemented *fsync* which checkpoint the entire system state to disk, while Linux only writes out the modified directory entry. There is a new *group sync*, that is not available under Linux, that only checkpoint the system state once at the end of each benchmark phase. *Group sync* guarantees that applications either run to completion or appears to have never started. In addition, it may afford some applications a significant speedup over Linux, as high as 200 factor [3].

For LFS large-file benchmark, there is three phases involved. For the first phase, a 100MB file was created by sequentially writing 8KB chunks, with one *fsync* call at the end. In this phase, HiStar achieves close to the maximum disk bandwidth of 58MB/sec. For the second phase, the algorithm tested random write throughput. Specifically, 100MB worth of 8KB chunks were written to random locations in a file, and the changes were *fsync* to disk for each 8KB write. For the third phase, the algorithm tested read performance by sequentially reading the 100MB file in 8KB chunks. This performance is comparable to Linux. However, since HiStar paged in the entire 100MB file when first accessed, thus the performance of random reads differs a little from the sequential case [3].

On top of all these benchmarks, HiStar also benchmarked on application level using GNU and GCC. In HiStar, most of the CPU time is spent in the user space, thus the performance is slower than Linux and comparable to OpenBSD. However, HiStar achieved good network througput when

they downloaded a 100MB file using *wget*. They also measured the time it took to check a 100MB file containing randomized binary data for viruses using ClamAV; HiStar performned comparably with Linux and OpenBSD [3].

7.2 Flume

When evaluating performance in Flume, the authors consider system security performance as well. Note that Flume is embedded in FlumeWiki, which is created by MoinMoin. Moin has a few bugs and they had shown that Flume was able to prevent these bugs from revealing private data. Furthermore, Flume's information flow control rules forced FlumeWiki to be built in such a way that does not leak information through its namespace [7].

With added security, there will of course be added overhead. Flume adds about 35-286μs of overhead to interposed system calls, which is a factor of 4 - 35. These overhead includes additional IPC, RPC marshalling, additional system calls, and extra computation for security checks. Also, most Flume system calls consist of two RPCs, from client application to reference monitor, to file server. This accounts for about 40μs of Flume's additional latency. For IPC overhead, it is due to the IPC proxies between processes. In standard Linux, IPC takes about four system calls: 2 reads and 2 writes. However, due to the proxies, Flume takes about 12 systems: 4 selects, 2 reads, and 2 writes [7].

To further evaluate the system level performance overhead of Flume, read and write experiments were designed. In the read experiments, a process requests pages from a pool of 200 pages, each is about 9 KB. In the write experiments, a process writes a 40 byte modification to one of the pages, of which a server responds with an 9 KB page. Overall, FlumeWiki is 43% slower than Moin in read and 34% slower in write throughput, thus this added about 40ms latency overhead [7]. This is probably because for each page read request, the reference monitor handles 753 system calls, which takes about 28 ms to handle.

Overall, Flume is slower than HiStar because based on the benchmarks HiStar performed comparably to Linux while Flume performed consistently worse than Linux.

8 Summary & Conclusion

There are many variants of Decentralized Information Flow Control (DFIC) operating systems. [9]. In this paper we have examined Flume and Histar, but there are also Luminar, Asbestos, DStar, DEFCon, Jflow, CQUAL, SELinux, and JIF [2]. Despite all the research there is still major work that needs to be done.

The major goal and the focus of future development of Decentralized Information Flow Control (DIFC) must lead to a more simplified system and also to be allowed to coexist with legacy software, both in the kernel and in the application level.

References

1. Roy, I., Porter, D.E., Bond, M.D., McKinley, K.S., Witchel, E.: Laminar: Practical fine-grained decentralized information flow control (2009)
2. Krohn, M., Tromer, E.: Non-interference for a practical difc-based operating system. In: in IEEE Symposium on Security and Privacy. IEEE Computer Society. (2009)
3. Zeldovich, N., Boyd-wickizer, S., Kohler, E., Mazieres, D.: Making information flow explicit in histar. In: In Proc. 7th OSDI. (2006)

4. Smith, J.W.: Inadequate security principles. (2012)
5. Liskov, A.C.M.B.: A decentralized model for information flow control. (1997)
6. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Frans, M., Eddie, K., Morris, K.R.: Information flow control for standard os abstractions. In: In SOSP. (2007)
7. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard os abstractions. SIGOPS Oper. Syst. Rev. **41**(6) (October 2007) 321–334
8. ClamAV: `http://www.clamav.net/`.
9. Albanesius, C.: Nov. 5 hacks target paypal, symantec, more. (2012)