Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory

Laurent Georget[1], Mathieu Jaume[2], Guillaume Piolle[1], Frédéric Tronel[1], and Valérie Viet Triem Tong[1]

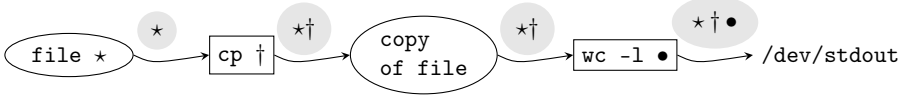
[1] EPC CIDRE CentraleSupelec/Inria/CNRS/Université de Rennes 1, Rennes, France
laurent.georget@irisa.fr
[2] Sorbonne Universités, UPMC, CNRS, LIP6 UMR 7606, Paris, France

Abstract. Information flow control can be used at the Operating System level to enforce restrictions on the diffusion of security-sensitive data. In Linux, information flow trackers are often implemented as Linux Security Modules. They can fail to monitor some indirect flows when flows occur concurrently and affect the same containers of information. Furthermore, they are not able to monitor the flows due to file mappings in memory and shared memory between processes. We first present two attacks to evade state-of-the-art LSM-based trackers. We then describe an approach, formally proved with Coq [12] to perform information flow tracking able to cope with concurrency and in-memory flows. We demonstrate its implementability and usefulness in Rfblare, a race condition-free version of the flow tracking done by KBlare [4].

Keywords: Information flow tracking; Linux; LSM

1 Introduction

Information Flow Control (IFC) at the Operating System (OS) scale is a security mechanism preventing leaks or improper manipulation of information stored in the system. At the OS level, flows are the actions of the processes causing the copy of data from a container of information to another. Containers are OS-level abstractions such as processes, files, message queues, network sockets, etc. The security status of each container is given by a security label, called a *taint*, initially set by a security officer. The taint of a container encodes the history of the flows that have altered its content. Consider the situation presented here:



The process *cp* copies a *file*, that the process *wc* reads to output the number of lines in it. *file* is originally tainted with ⋆, *cp* with †, *wc* with •. IFC maintains the knowledge of past flows in the system through *taint propagation*. When a flow occurs, the taint of the destination is updated with the taint of the source, in order to record the flow in the system. Assuming flows are performed from

left to right in the example, we see that the taint from the *file* is propagated to *cp* and then to the *copy* and *wc*, and eventually to */dev/stdout*. The focus of this article is on three IFC systems developed for the generic Linux kernel: KBlare [4], Laminar [11, 9], and the Android Linux kernel: Weir [8].

The implementations of Laminar, KBlare, and Weir are based on the Linux Security Modules (LSM) framework. This framework provides (1) extra security fields in Linux's internal data structures and (2) a set of callbacks, called LSM *hooks*, positioned in the code of system calls. Security mechanisms can register functions on these hooks to be executed just before security-sensitive operations are made, to enforce security decisions. The crucial property we expect from these systems is that they are able to correctly track all flows in the system. If malware could escape them, the confidentiality and integrity of user data would no longer be guaranteed.

However, the LSM framework has been conceived primarily for access control, and not for IFC [14] so it is not obvious that it is also appropriate for this purpose. In a previous work [3], we have developed an approach to verify that LSM hooks were available in each system call generating an information flow, so that an information flow tracker could monitor all of them. This is a necessary condition to implement a correct information flow tracker. This approach led to the identification of some shortcomings in the placement of LSM hooks and the addition of a few hooks. This necessary condition is nevertheless not a *sufficient* one. Indeed, detecting each individual direct flow is not equivalent to detecting *all* flows. Flows occurring concurrently in the system and involving a common information container may cause *indirect* flows (i.e. compositions of individually detected flows) to be undetected. This is because the sequence ⟨detection of the flow by the tracker (a function registered in a LSM hook) ; actual occurrence of the flow (another function called later in the same system call)⟩ is not atomic, and thus taint propagation is subject to *race conditions*.

Information flow trackers also have to cope with the existence of *continuous flows*, which are started by one system call and stopped by another. A typical example of these flows is caused by shared memory segments. When a process shares a memory segment with another one, they can freely communicate through it. No system call is required for a process to read and write its own memory, and thus, the trackers cannot see these individual flows. Therefore, they have to make the overapproximation that the flow is occurring "continuously" between the system call setting the shared memory up and the system call shutting it down. Even if a security mechanism could tolerate missing some flows, the hassle of handling race conditions is justified by the existence of these *continuous flows*, which cannot be monitored otherwise. Comments in the source code of Blare and Laminar stress the importance of the issue and the lack of a straightforward solution. In this article, we propose a solution to handle concurrency between system calls as well as continuous flows. To the best of our knowledge, our approach is the first to propose a provably correct way to do so.

We propose three contributions in this article, described in Sections 2 to 4. The paper is organized as follows.

In Section 2, we detail two ways of evading Laminar, KBlare, and Weir by exploiting a race condition between a read and a write operations and by exploiting indirect continuous flows between files mapped in memory.

In Section 3, we describe formally the mechanism of taint propagation common to the three tools as well as the concrete flows of information between containers. We describe how considering flows as operations spanning over some time instead of atomic ones allow us to propose a new way of propagating taints that takes into account all possible flows in the system. We prove two properties on our result: (1) the overapproximation of flows we compute is sound, no flow can be missed; (2) the overapproximation is the smallest one in our model, i.e. any smaller overapproximation would be unsound because there would exist a way to perform an indirect flow missed by the algorithm. We have proved the correctness of our algorithm in our model with the Coq proof assistant [12].

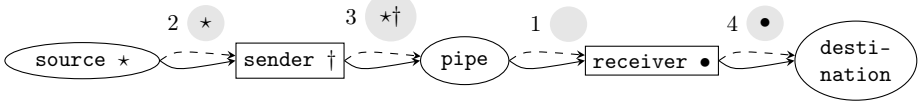
In Section 4, we describe Rfblare, a new taint propagation mechanism for KBlare, implementing our solution from the previous point. We show that it correctly handles the attacks developed in the first point and encurs little overhead. This work describes precisely how the LSM framework can be modified to allow the implementation of correct flow tracking, independently from any particular semantics for the labels or any flow policy model.

We revise related works in Section 5 and conclude in Section 6. Due to space limitations, we give in this article an overview of our main results. Our complete implementation, tests, as well as the proofs of our formal results are available online on the project's website: `https://blare-ids.org/rfblare`.

2 Evading Existing Information Flow Trackers

2.1 Exploiting a race condition to copy a file without its taint

KBlare, Laminar and Weir use the `file_permission` LSM hook when a process reads from or writes to a file to perform the taint propagation. There is a risk of race conditions even on uniprocessor systems because the process is allowed to sleep and yield the CPU between the time the LSM hook is triggered and the time the function that actually performs the flow is called. This can be exploited to copy a file without its taint. Consider this example:



The dashed lines represent the observations of flows by the tracker (made in the order given by the numbering for the sake of this example) and the plain lines the actual flows in the system (done from left to right). The content from the *source* file is propagated to the *destination* via processes *sender* and *receiver* and the pipe. Nevertheless, since *receiver* has started reading from the pipe before *sender* has written to it, the taint is not propagated properly.

This attack works reliably and effectively on KBlare as a simple one-liner: `mkfifo pipe; cat < pipe > destination& cat < source > pipe`. On Weir,

it is more difficult because processes benefit from a stricter isolation by default. To perform the attack, we have developed two toy applications. One offers a text box in which the user can write a message. When clicking on a `send` button, the process allocates a new security tag from the Weir tag manager and then copy this message to a pipe whose name is known by both applications. The other application has a text area and a `receive` button. Clicking on the latter copies the message from the pipe to the text area. We install both applications with the same user id so that they can share a folder, and we create the pipe inside. We observe that if the user clicks on `send` before clicking on `receive`, the logs from Weir show that the taint is correctly propagated to the *receiver*. However, clicking on `receive` first triggers the race condition. The apparent result is the same (the text appears in the *receiver*'s text area) but the logs show that the taint from the *sender* only reaches the *pipe* and not the *receiver*.

On Laminar, this attack is not possible because the developers put the entire system call in a critical section to prevent race conditions between concurrent reads and writes. However, this solution is not entirely satisfactory. First of all, it incurs sacrificing parallelism on very common system calls from the family of `read` and `write`. Reading a file from a high-latency filesystem (e.g. a network-backed filesystem) might block all other reads and writes for a long time. Secondly, many existing applications rely on the semantics that reading from an empty pipe blocks until it is written to. This is not possible if operations cannot interleave. Laminar does not care about this since it mandates that all pipes are open in non-blocking mode to avoid a covert channel but KBlare and Weir are committed not to require any porting of existing applications.

2.2 Making a flow in memory

Ordinarily, to read from and write to a file, a process uses system calls from the `read` and `write` family but this is not the only way. There exists a system call, `mmap`, which can make a file (or better said, the pages of cache memory buffering the file's content) appear as part of the process's memory space. When a file is mapped in its memory, a process can read from it and write to it (provided that the mapping has the read–write permissions) without system calls, by usual memory manipulations. Another system call, `munmap`, unmaps a file. The same mechanism is used to share memory segments between processes. If two processes map the same file, usually a temporary anonymous file, in a read–write public way, they can communicate and collaborate on the same data.

Weir does not handle mappings. Laminar has an interesting comment in its source code stating: "XXX: Should do something about mmaped files." [10, file `security/difc.c`, l. 944] which suggests that the problem is known to the developers and considered important enough but is not trivially solvable. KBlare supports propagating taints between the file and the memory space of the process when the former is mapped. It also claims to handle shared memory segments by maintaining a list of attached shared memory segments for each process. When propagating taints from or to this process, the shared memory segments participate in the propagation [4]. The implementation is incomplete however

and even the design is somewhat flawed. If processes A and B share memory, as well as processes B and C, we must consider that processes A and C share memory, although they do so indirectly. KBlare fails to handle that "transitivity".

We can reuse the attack from the previous section by replacing the reading by *sender* from *source* by a read-only mapping, the writing by *receiver* to *destination* by a read-write mapping and finally, the *pipe* by a shared memory segment between *receiver* and *sender*. We could not test it on KBlare, Laminar and Weir directly because of the lack of implementation on these platforms but a toy implementation of KBlare's described propagation [4] shows the problem: depending on the order in which the mappings and the shared memory are set up, taints are not propagated from *source* to *destination* in all cases, although the content from *source* is copied to *destination*.

These two attacks show that the trackers' observations are inconsistent with the actual flows altering the containers. A sound tracker needs at least to compute an overapproximation of the actual flows. To solve this problem, we propose in the following section a formal model of the propagation done by the trackers, what a perfect propagation would be and a way to compute a correct overapproximation of this perfect propagation in practice.

3 A New Algorithm for Taint Propagation

We propose a formal model of information flows between containers as well as a formal description of taint propagation in order to describe the shortcomings of current trackers and prove the correctness of our taint propagation.

3.1 Tags, Information Flows and Executions

A container is anything in the system that can carry data (usually originating from a user). Files, network sockets, pipes are examples of containers. We write $\mathcal{C}$ for the set of containers of information. Contrarily to most approaches, we do not consider processes or threads as containers of information *per se*. Instead, we consider that the *memory space* of each process is a container (i.e. a flow from/to a process is a flow from/to its memory). This distinction is useful since in Linux systems, it is possible to create distinct processes sharing their entire memory space, and thus the data they store and produce.

To record the origin of the information stored in any container, a tracker attaches a security label, also called a taint, to all of them. In our model, a taint is a set of *tags*. In the example above, we have chosen the set $\{\star, \dagger, \bullet\}$ as tags whereas actual implementations generally use a predefined range of integers. Without loss of generality, we consider that each container $c \in \mathcal{C}$ is initially associated to a unique tag written t^c. Intuitively, a tag represents a primary source of information in the system. Let $\mathcal{T} = \uplus_{c \in \mathcal{C}} \{t^c\}$ be the set of all tags. During the lifetime of the system, as information gets exchanged between containers, the tracker's task is to reflect these changes in the set of tags associated to the containers. This is the taint propagation.

Definition 1 (Configuration, Taint). *A configuration* $\theta : \mathcal{C} \to \wp(\mathcal{T})$ *maps each container to its set of tags.* $\theta(c) = \{t^{c_1}, \ldots, t^{c_n}\}$ *is called the* taint *of* c *and indicates that* c *contains information originating from containers* c_1, ..., c_n.

We write Θ for the set of configurations and θ_{init} for the initial configuration such that $\forall c \in \mathcal{C}\ \theta_{init}(c) = \{t^c\}$. A configuration is an abstraction of the state of the containers, it represents an overapproximation of the sources of information contributing to the current content of a container. This state evolves upon occurrence of specific events relative to information flows. An information flow is the copy of (a portion of) the content of a container, called the *source* of the flow, into another, called the *destination*. We consider that an information flow is not an atomic operation. Instead, we consider that a flow is successively *enabled*, *executed* and *disabled*. The execution of the flow (i.e. the copy of information) may happen only after it is enabled, and before it is disabled. It may happen once, several times, or even not at all. Several different flows from a container c_1 to a container c_2 may occur during the lifetime of the system, and may even overlap. In order to distinguish them, we introduce the set $\mathcal{F}$ of flow identifiers (typically, we choose $\mathcal{F} = \mathbb{N}$ so that each flow is uniquely identified by a ever-increasing counter).

Definition 2 (Event). *Let* $c_1, c_2 \in \mathcal{C}$ *and* $f \in \mathcal{F}$. *We define the relation* $c_1 \to_f c_2$ *which is to be understood as a flow called* f *from* c_1 *to* c_2. *An event* $e \in \mathcal{E}$ *is either a pair* $(f, (c_1, c_2))$ *where* $c_1 \xrightarrow{enable}_f c_2$ *or* $c_1 \xrightarrow{disable}_f c_2$, *or a pair* $(f, (c_1, c_2))$ *where* $c_1 \xrightarrow{exec}_f c_2$. *We call the first set* $\mathcal{O}$ *and the second one* $\mathcal{X}$. *These relations have the following intuitive meaning:*

$c_1 \xrightarrow{enable}_f c_2$ *means that the flow named* f *from* c_1 *to* c_2 *is enabled*
$c_1 \xrightarrow{exec}_f c_2$ *means that the flow named* f *from* c_1 *to* c_2 *is executed*
$c_1 \xrightarrow{disable}_f c_2$ *means that the flow named* f *from* c_1 *to* c_2 *is disabled*

In other words, $\mathcal{O}$ contains the events enabling and disabling flows whereas $\mathcal{X}$ contains the events corresponding to actual flow executions.

We write $\mathbf{E} \subseteq \mathcal{E}^+$ for the set of executions, defined as non-empty sequences of events. We write $e[i]$ for the i-th event of an execution $e \in \mathbf{E}$, $\lg(e)$ the length of e and $e[:n]$ (resp. $e[n:]$) the prefix $(e[1], \ldots, e[n])$ of length n of e (resp. the suffix $(e[n], \ldots, e[\lg(e)])$ of length $\lg(e) - n + 1$ of e). Executions in $\mathbf{E}$ satisfy two conditions of causality: a flow is always enabled before being executed or disabled, and cannot be executed after it is disabled:

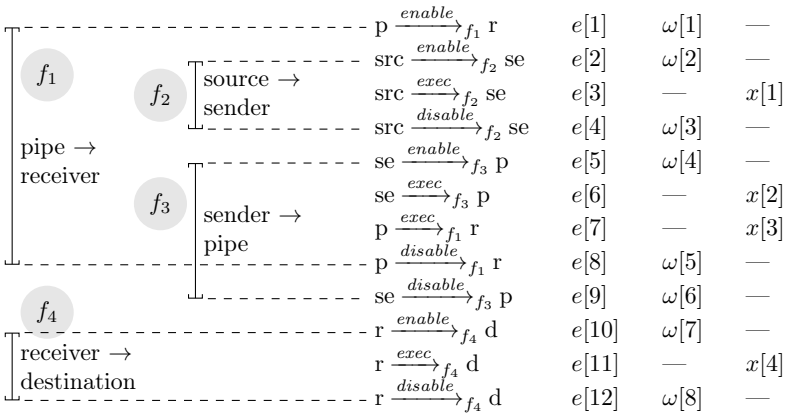
$$\forall i\ \ e[i] = c_1 \xrightarrow{disable}_f c_2 \vee e[i] = c_1 \xrightarrow{exec}_f c_2 \Rightarrow$$
$$\left(\exists j < i\ \ e[j] = c_1 \xrightarrow{enable}_f c_2 \wedge \left(\forall k\ j < k < i \Rightarrow (e[k] \neq c_1 \xrightarrow{disable}_f c_2)\right)\right) \quad (1)$$

We suppose that only events in $\mathcal{O}$ are observable by the tracker and that it cannot react on events in $\mathcal{X}$. This models the fact that a tracker cannot perform taint propagation all the time during the execution of a system call but only when the execution reaches a LSM hook. We write $\mathbf{O}$ the set of observable executions, containing only events in $\mathcal{O}$, and $\mathbf{X}$ the set of concrete executions, containing only

events in $\mathcal{X}$. Observable executions are the sequences of events observed by the tracker whereas concrete executions describe the executions of flows, i.e. how the content of containers actually changes over time. Given $e \in \mathbf{E}$, we write $e_{\mathcal{O}} \in \mathbf{O}$ (resp. $e_{\mathcal{X}} \in \mathbf{X}$) the observable execution (resp. the concrete execution) obtained by removing the unobservable events (resp. the observable events) from e. We define a compatibility relation between observable and concrete executions.

Definition 3 (Compatibility). *An observable execution ω is compatible with a concrete execution x iff they are projections from an execution in $\mathbf{E}$. Formally, $\forall x \in \mathbf{X}\ \forall \omega \in \mathbf{O}$, we write $\omega \vdash x$ iff $\exists e \in \mathbf{E}\ (\omega = e_{\mathcal{O}} \wedge x = e_{\mathcal{X}})$.*

Example 1. We consider the first attack presented in Section 2 and illustrate it on the figure below. We abbreviate the name of the containers in the rest of this article: *src* is the source, *se* the sender, p the pipe, r the receiver and d the destination. The x column represents the concrete execution of flows between the containers of information. The ω column is the sequence of enabling and disabling events seen by the tracker. These two executions are compatible because there exists an execution e which is a linearization of both, respecting the causality conditions expressed by (1).



3.2 Flow-based Interpretations of Executions

A tracker is said to be *sound* if it does not miss any flow. However, a given observable execution can correspond to several concrete executions, when several flows are enabled at the same time. Thus, a tracker cannot track flows with an absolute precision. Thus, a sound tracker can only provide an overapproximation of the taints considering the flows generated by all the compatible concrete executions. Actually, in the example, due to the synchronous nature of the pipe, there is only one possible execution order between the reading and the writing because reading from the pipe blocks until some content has been written to it. We chose a pipe because it makes triggering the race condition trivial but we could replace the pipe by a regular file in the example, in which case the order of execution would not be constrained.

Ideal Tag Propagation We define a transition relation $\hookrightarrow \subseteq \Theta \times \mathcal{X} \times \Theta$ describing how the information flows influence the content of containers.[3]

$$\forall \theta, \theta' \in \Theta\ \forall c_1 \xrightarrow{exec}_f c_2 \in \mathbf{X}\ \theta \xhookrightarrow{c_1 \xrightarrow{exec}_f c_2} \theta' \leftrightarrow \theta' = \theta[c_2 \leftarrow \theta(c_2) \cup \theta(c_1)]$$

For $x \in \mathbf{X}$ we write $\theta_0 \xhookrightarrow{x[:n]} \theta_n$ when $\theta_0 \xhookrightarrow{x[1]} \theta_1 \xhookrightarrow{x[2]} \cdots \theta_{n-1} \xhookrightarrow{x[n]} \theta_n$. This relation is the way an ideal tracker would propagate tags if it could observe the execution of the flows themselves instead of the enabling and disabling events. Table 1a details the tag propagation represented by this relation according to the concrete execution x from Example 1.

Tag Propagation by LSM-based Trackers Formally, the computation done by LSM-based trackers such as Laminar, KBlare, and Weir can be described by a transition relation $\rightharpoonup \subseteq \Theta \times \mathcal{O} \times \Theta$ defined as follows:

$$\theta \xrightarrow{c_1 \xrightarrow{enable}_f c_2} \theta[c_2 \leftarrow \theta(c_2) \cup \theta(c_1)] \qquad \theta \xrightarrow{c_1 \xrightarrow{disable}_f c_2} \theta$$

Considering again Example 1, Table 1b describes the computation done from ω. This taint propagation is not sound: it can miss indirect flows. For example, in the concrete execution x compatible with ω, there is an indirect flow from *source* to *destination* ($t^{\text{src}} = \bigstar \in \theta(\text{d}) = \{t^{\text{d}} = \blacktriangle, t^{\text{r}} = \circledcirc, t^{\text{p}} = \square, t^{\text{se}} = \blacksquare, t^{\text{src}} = \bigstar\}$ in the computation made by $\hookrightarrow$), but this is not the case in the computation made by $\rightharpoonup$.

This model describes straightforwardly "floating labels" systems such as Blare and Weir, in which a flow automatically updates the label of the destination container with the label of the source to show the dissemination of the tagged data. It also describes correctly, although this is less intuitive, the behavior of systems such as Laminar in which labels must be changed explicitly by the process. In both cases, the race condition is the same and has the same effect. In Blare and Weir, the flow occurs but the label of the destination is not updated accordingly. If this flow is illegal, then the violation of the security policy is not detected. In Laminar, even if the flow is illegal, which means that the destination label does not dominate the source one, no alert is raised and it occurs anyway. In our model, the labels only represent the knowledge the tracking system has about past flows in the system, and is not tied to any specific policy semantics.

Computation of the Smallest Correct Overapproximation Given an observable execution ω, we define $\mathit{Enabled}_\omega \subseteq \mathcal{C} \times \mathcal{C}$ as the set of flows that have been enabled during ω and not disabled (yet) at the end of ω. $\mathit{Enabled}^*_\omega$ stands for the reflexive and transitive closure of relation $\mathit{Enabled}_\omega$.

$$(c_1, c_2) \in \mathit{Enabled}_\omega \Leftrightarrow \exists i\ \omega[i] = c_1 \xrightarrow{enable}_f c_2 \wedge \forall j > i\ \omega[j] \neq c_1 \xrightarrow{disable}_f c_2$$

An overapproximation, written $\mathit{Flows}_\omega \subseteq \mathcal{C} \times \mathcal{C}$, of flows that can be generated by some concrete execution compatible with a given observable execution $\omega \in \mathbf{O}$

[3] $f[x \leftarrow a]$ is the function such that $f[x \leftarrow a](y) = \begin{cases} a & \text{if } x = y \\ f(y) & \text{otherwise.} \end{cases}$

can be computed as follows.[4]

$$\mathit{Flows}_\omega = \begin{cases} \mathit{Enabled}^*_\omega & \text{if } \lg(\omega) = 1 \\ \mathit{Flows}_{\omega[:k]} \,.\, \mathit{Enabled}^*_\omega & \text{if } \lg(\omega) = k+1 \end{cases}$$

For example, if the flow (A, B) has happened in the past, and the flow (B, C) gets enabled, then the composition (A, C) is a new flow in the system. This would not be the case if the flow (B, C) were anterior to (A, B). Considering Example 1, Table 1c illustrates how Flows_ω is computed. As we can see, Flows_ω is not necessarily a transitive relation. Proposition 1 below ensures the soundness of the tag propagation mechanism, as illustrated in Table 1d. Proposition 2 ensures that it is impossible to compute a better overapproximation in our model.

Proposition 1 (Soundness). *Flows generated by a concrete execution compatible with an observable execution ω belong to Flows_ω.*

$$\forall e \in \mathbf{E}\ \forall \theta \in \Theta\ \theta_{init} \xrightarrow{e_\mathcal{X}} \theta \Rightarrow \forall c \in \mathcal{C}\ \theta(c) \subseteq \bigcup_{(c',c) \in \mathit{Flows}_{e_\mathcal{O}}} \theta_{init}(c')$$

Proof (Sketch). By induction on $\lg(e)$. It suffices to show that if a concrete execution exists, then, by the causality conditions, there necessarily exists a sequence of observable events that have enabled the flows executed in the concrete execution. Flows_ω contains these flows by construction.

Proposition 2 (Smallest overapproximation / Completeness). *All flows in Flows_ω are generated by at least one concrete execution compatible with the observable execution ω.*

$$\forall \omega \in \mathbf{O}\ \forall c, c' \in \mathcal{C}$$
$$(c, c') \in \mathit{Flows}_\omega \Rightarrow \exists x \in \mathbf{X} \left(\omega \vdash x \wedge \forall \theta \in \Theta\ \theta_{init} \xrightarrow{x} \theta \Rightarrow \theta_{init}(c) \subseteq \theta(c') \right)$$

Proof (Sketch). By induction on $\lg(\omega)$. Suppose that $(c, c') \in \mathit{Flows}_{\omega[:n]}$ is the flow $(c = c_1, c_2), (c_2, c_3), \ldots, (c_{m-1}, c_m = c')$. Then by definition, there exists $i \leq m$ such that $(c_1, c_i) \in \mathit{Flows}_{\omega[:n-1]}$ and $(c_i, c_m) \in \mathit{Enabled}^*_{\omega[:n]}$. By the induction hypothesis, there exists $x \vdash \omega[:n-1]$ which propagates tags from c_1 to c_i. Concatenating x with the executions of the flows $(c_i, c_{i+1}), \ldots, (c_{m-1}, c_m)$ (which are enabled and not disabled yet in $\omega[:n]$) in this order yields a concrete execution $x' \vdash \omega[:n]$ propagating tags from $c = c_1$ to $c_m = c'$ via c_i.

4 Implementation and Experiments

We have implemented our taint propagation algorithm as Rfblare, the *race-free KBlare*, into the version 4.7 of the vanilla Linux kernel. We have not contributed to the policy enforcement part of KBlare and do not discuss it here. Rfblare covers the flows listed in Table 2. Consistently with the formal description of our algorithm, we use one LSM hook as an enabling event and another one as a

[4] Given two relations $R_1 \subseteq E \times F$ and $R_2 \subseteq F \times G$, the relation $R_1 \,.\, R_2 \subseteq E \times G$ is defined by $(x, y) \in R_1 . R_2$ iff there exists $z \in F$ such that $(x, z) \in R_1$ and $(z, y) \in R_2$.

Table 1. Flow-based Interpretations of Executions. For the sake of legibility, we note: $t^{src} = \bigstar$, $t^{se} = \blacksquare$, $t^{p} = \square$, $t^{r} = \circledcirc$, $t^{d} = \blacktriangle$

(a) Computation of $\theta_{init} \xhookrightarrow{x[:n]} \theta$ (Ideal propagation)

n	$x[n]$	$\theta(src)$	$\theta(se)$	$\theta(p)$	$\theta(r)$	$\theta(d)$
1	$src \xrightarrow{exec}_{f_1} se$	$\bigstar$	$\blacksquare, \bigstar$	$\square$	$\circledcirc$	$\blacktriangle$
2	$se \xrightarrow{exec}_{f_2} p$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc$	$\blacktriangle$
3	$p \xrightarrow{exec}_{f_3} r$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square, \blacksquare, \bigstar$	$\blacktriangle$
4	$r \xrightarrow{exec}_{f_4} d$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square, \blacksquare, \bigstar$	$\blacktriangle, \circledcirc, \square, \blacksquare, \bigstar$

(b) Computation of $\theta_{init} \xrightarrow{\omega[:n]} \theta$ (LSM-based trackers)

n	$\omega[n]$	$\theta(src)$	$\theta(se)$	$\theta(p)$	$\theta(r)$	$\theta(d)$
1	$p \xrightarrow{enable}_{f_1} r$	$\bigstar$	$\blacksquare$	$\square$	$\circledcirc, \square$	$\blacktriangle$
2	$src \xrightarrow{enable}_{f_2} se$	$\bigstar$	$\blacksquare, \bigstar$	$\square$	$\circledcirc, \square$	$\blacktriangle$
4	$se \xrightarrow{enable}_{f_3} p$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square$	$\blacktriangle$
7	$r \xrightarrow{enable}_{f_4} d$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square$	$\blacktriangle, \circledcirc, \square$

(c) $Enabled_{\omega[:n]}$, $Enabled^{*}_{\omega[:n]}$ and $Flows_{\omega[:n]}$

n	$Enabled_{\omega[:n]}$	$Enabled^{*}_{\omega[:n]}$	$Flows_{\omega[:n]}$
0		(src, src), (se, se), (p, p), (r, r), (d, d)	(src, src), (se, se), (p, p), (r, r), (d, d)
1	(p, r)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r)
		...	
7	(r, d)	(src, src), (se, se), (p, p), (r, r), (d, d), (r, d)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r), (p, d), (src, se), (src, p), (src, r), (src, d), (se, p), (se, r), (se, d)

(d) Computation of $\bigcup_{(c_1, c_2) \in Flows_{\omega[:n]}} \theta_{init}(c_1)$ (Rfblare's Overapproximation)

n	$\omega[n]$	$\theta(src)$	$\theta(se)$	$\theta(p)$	$\theta(r)$	$\theta(d)$
1	$p \xrightarrow{enable}_{f_1} r$	$\bigstar$	$\blacksquare$	$\square$	$\circledcirc, \square$	$\blacktriangle$
2	$src \xrightarrow{enable}_{f_2} se$	$\bigstar$	$\blacksquare, \bigstar$	$\square$	$\circledcirc, \square$	$\blacktriangle$
3	$src \xrightarrow{disable}_{f_2} se$	$\bigstar$	$\blacksquare, \bigstar$	$\square$	$\circledcirc, \square$	$\blacktriangle$
4	$se \xrightarrow{enable}_{f_3} p$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square, \blacksquare, \bigstar$	$\blacktriangle$
5	$p \xrightarrow{disable}_{f_1} r$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square, \blacksquare, \bigstar$	$\blacktriangle$
6	$se \xrightarrow{disable}_{f_3} p$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square, \blacksquare, \bigstar$	$\blacktriangle$
7	$r \xrightarrow{enable}_{f_4} d$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square, \blacksquare, \bigstar$	$\blacktriangle, \circledcirc, \square, \blacksquare, \bigstar$
8	$r \xrightarrow{disable}_{f_4} d$	$\bigstar$	$\blacksquare, \bigstar$	$\square, \blacksquare, \bigstar$	$\circledcirc, \square, \blacksquare, \bigstar$	$\blacktriangle, \circledcirc, \square, \blacksquare, \bigstar$

disabling event for each flow. We have leveraged the expertise from our previous work on LSM [3] to map our model onto the LSM framework. Some flows cannot possibly enter in a race condition with others and require no disabling hook. For example, the `execve` system call is used to run a new program, causing a flow from the executable file to the memory of the process. However, this flow cannot race with any flow to the file, because it is forbidden both to write into a file being executed and to execute a file being written to. In the case of `fork`, no race condition occurs with the new process since it has not started yet, and race conditions with the parent process are irrelevant since the `mm_dup_security` hook is actually called *after* the copy of the parent process's memory is finished (i.e. after the flow has taken place). `mq_timedsend` and `msgsnd` are in the same situation. When a message is to be sent to a message queue, it is first copied to a buffer in the kernel, then checked by the LSM module before it can be registered to the queue. This order of actions prevents the calling process from tampering with the message being checked. Since the kernel already avoids the data race condition on the message, using our algorithm would be redundant.

We have added two LSM hooks as disabling events: `syscall_before_return` for discrete flows and `ptrace_unlink` for `process_vm_readv` (discrete flow) and `ptrace` (continuous flow). `ptrace` lets one process attach to another and monitor its execution. It is used by debuggers. We consider it a continuous flow because it opens many ways for the tracer process to exchange data with the tracee, in overt or covert ways. We have placed the `syscall_before_return` hook before the normal return of the system calls generating the flows. The case of `mmap` and `mprotect` is special. We need not track the unmapping of files because the kernel already does so. More precisely, for any file, we can query the kernel for the list of processes' memory spaces it is mapped into, and for any memory space, we can similarly know which files are mapped into. Therefore, when computing the taint propagation for a flow to a file or a memory space, we use this knowledge maintained by the kernel to take into account the continuous flow caused by the mapping. We still need the enabling hooks nonetheless to perform the taint propagation between the file and the memory space as soon as the file is mapped. If the mapping is read-only, the flow is from the file to the memory space, otherwise, it is bidirectional. `mprotect` can be used to change a read-only mapping to a read-write one, so we need to monitor it.

We have tested the attacks presented in Section 2. In the case of the *Stealthily Copying a File*, the sequence of events is of course still the same but Rfblare reacts correctly to it. The flow from the pipe to the receiver is *enabled* when the receiver goes through the *file_ permission* hook. The flow remains enabled as the process is blocked for the pipe to be written to. On the sender side, the flow from the source file to the sender is enabled (the source's tags are propagated to the sender), and then disabled immediately after. Then, when the sender process writes to the pipe, the flow from the sender to the pipe is enabled and the tags from the sender (which includes tags from the source file) are propagated to the pipe. Since the flow from the pipe to the receiver is still enabled, the tags are also propagated to the receiver. Finally, when the receiver restarts, the actual

Table 2. Flows monitored by Rfblare

System call family	Flow	Enabling event	Disabling event
Discrete flows			
`read`	File→memory	`file_permission`[a]	`before_return`[b]
`write`	Memory→file	`file_permission`[a]	`before_return`[b]
`recv`	Socket→memory	`socket_recvmsg`[a]	`before_return`[b]
`send`	Memory→socket	`socket_sendmsg`[a]	`before_return`[b]
`process_vm_readv`	Memory→memory	`ptrace_access_check`[a]	`ptrace_unlink`[b]
`migrate_pages`	Memory→memory	`task_movememory`[a]	`before_return`[b]
`move_pages`	Memory→memory	`task_movememory`[a]	`before_return`[b]
`msgrcv`	Message queue → memory	`mq_store_msg`[a]	`before_return`[b]
`msgsnd`	Memory → message queue	`msg_msg_alloc_security`[a]	—[c]
`mq_timedreceive`	Message queue → memory	`mq_store_msg`[a]	`before_return`[b]
`mq_timedsend`	Memory → message queue	`msg_msg_alloc_security`[a]	—[c]
`clone`/`fork`	Memory→memory	`mm_dup_security`[b]	—[c]
`execve`	File→memory	`bprm_set_creds`/ `bprm_committing_creds`[a]	—[c]
`kill`	Memory→memory	`task_kill`[a]	`before_return`[b]
Continuous flows			
`mmap`	File↔memory	`mmap_file`[a]	—[d]
`mprotect`	File↔memory	`file_mprotect`[a]	—[d]
`ptrace`	Memory↔memory	`ptrace_access_check`[a]	`ptrace_unlink`[b]
`ptrace`	Memory↔memory	`ptrace_traceme`[a]	`ptrace_unlink`[b]

[a] LSM hook already present in the LSM framework.
[b] LSM hook added by us.
[c] No LSM hook needed because this operation cannot race with any other.
[d] No LSM hook needed because we query the kernel for the active mappings.

Table 3. Linux compilation micro-benchmark results. Times are given as an average over thirty runs, with the 95% confidence interval. Ratios are the fraction of each system time over the reference system time and the 0-tags system time, respectively.

number of tags	user time (s)	system time (s) and ratios			elapsed time (s)
(reference)	1180 ±10.8	82.95 ±0.75	1.000	0.981	170.8 ±1.7
0	1174 ± 8.4	84.56 ±0.46	1.019	1.000	170.1 ±1.3
400	1175 ±10.3	84.66 ±0.55	1.021	1.001	170.8 ±1.5
800	1175 ±10.6	84.82 ±0.57	1.022	1.003	170.1 ±1.5
1200	1173 ±10.2	84.90 ±0.58	1.023	1.004	170.9 ±1.5
1600	1169 ±10.2	86.43 ±1.43	1.042	1.022	171.3 ±1.8
2000	1168 ± 9.5	86.92 ±1.58	1.048	1.027	170.6 ±1.8

reading is performed, the flow from the pipe to the receiver is disabled and the `read` system call finishes. The receiver then writes to the destination file, and thus propagates its taint to it. The content of the destination file is correctly reflected by its taint, despite the flows involving the pipe having occurred in the reverse order with respect to the corresponding passages through the *file_permission* hook. For the second attack with memory-mapped files and shared memories, we have used a similar setup. The sender and the receiver map respectively the source and destination file, and the pipe is replaced by a shared memory segment. We observe again the correct behavior: whichever mapping is done last (either one of the file, or the shared memory), the tags of the source file are propagated to all containers linked by the enabled continuous flows.

Measuring the overhead caused by Rfblare is critical to ensure its practicality. Our testcase is a compilation of the Linux kernel, version 4.7, on a machine with Rfblare. We place a unique tag on a varying number of source files to study the impact of the number of tags to propagate on the performance on the kernel. We believe compilation to be an appropriate benchmark because it is reproducible reliably and involves numerous flows to and from files as well as the spawning of numerous processes. Furthermore, it is relatively easy to verify the correct propagation because we put a unique tag on each source file and we can check by other means what files are supposed to participate to the compilation of each intermediary and final output of the compiler. Our results are presented in Table 3. We measure the time taken by the compilation depending on the number of tagged source files. As a reference, we have taken the time on a similar system without Rfblare. Tests are run thirty times each, on a virtual machine with 16Gb of RAM, and 8x3.2Ghz CPUs. The user time is the cumulated time spent by all threads outside the kernel. Logically, it shows no significant variation. The system time is the cumulated time spent in the kernel doing system calls, including taint propagation. Overall, on the Rfblare-equipped system, there is an increase of about 2 to 5% of the system time. This is small, especially if we consider the wall clock time spent during the compilation (column "elapsed time") which shows no significant variation.

5 Related Work

IFC has been an active topic of research and prototyping for a long time. It can be applied in programming languages or at the OS level, we only discuss the latter case here. Along with the various implementations, formal descriptions have been proposed, following the seminal work of Denning [2]. Denning showed that information flow policies could be described as lattices of security labels. This works helped reasoning about the respective expressiveness and objectives of the different kinds of policies. However, Denning only describes *access control* policies. The difference between access control and information flow control is explicited by Jaume et al. [5]: IFC bases its security decisions based on the history of flows in the system (maintained with taint propagation) whereas access control does not maintain this knowledge. The practical consequence is that thanks to

this knowledge, IFC allows more policies while maintaining the guarantee that no illegal flow can occur. For example, it is possible to let a process read a secret file or communicate with an unauthorized process, but not both. Access control can either allow both (which is a security hazard) or deny both (which is overly restrictive). However, despite the extensive literature on the models of labels (for example: [13, 15]) and on the properties enforceable with IFC, like non-interference [6], there is little formal work on taint propagation itself. However, implementing IFC in Linux systems raises practical difficulties, mainly due to concurrency and arcane corner cases in both the design and the implementation of the Linux kernel. Flume [7] is an IFC system implemented as an execution monitor in userspace, able to track the flows done by an individual process. It uses a LSM module to propagate taints to and from files. This is different from our solution, implemented entirely in-kernel which tracks flows in the entire system. Flowx [1] is a LSM module enforcing non-interference in an entire Linux system. Its implementation covers all IPCs present in Linux systems, including shared memory. However, it does not perform IFC according to our definition but rather access control since it does not maintain a knowledge of the flows in the system. Instead, it dynamically instantiates copies of existing containers of information with appropriate labels of security, each time an illegal access is asked for. We have already discussed the case of KBlare [4], Laminar [11] and Weir [8], which have a similar design. The main differences are the target (Android for Weir, all Linux systems for Laminar and KBlare), the model of label (inherited from Flume [7] for Laminar and Weir, radically different for KBlare) and the use of floating labels (KBlare and Weir) versus explicit changes (Laminar). They claim to cover a different range of overt and covert channels of information, Laminar putting a special focus on covert channels while KBlare disregarding them completely.

6 Conclusion

Information flow trackers are powerful tools to maintain a history of how data is disseminated and used in an operating system. This knowledge is necessary to enforce strong information flow policies or analyze malware activity. In Linux, most trackers are implemented using the Linux Security Modules framework, which provides hooks trackers can use to monitor the system calls making information flow. However, being able to monitor individual flows is not a guarantee of being able to correctly trace them all. We have shown that information flows generated by concurrent system calls can cause trackers to miss indirect information flows because of race conditions. To handle this issue, we have modeled information trackers as being able to monitor not the execution of flows themselves, but rather the events that enable and disable the flow. With this model as a basis, we have designed and proved an algorithm to compute the smallest overapproximation of the flow tracking in a given execution, considering all sequences of flow executions compatible with the events observable in this execution. The solution we propose has the very practical consequence that it makes possible to track contin-

uous flows, including continuous flows caused by memory mappings and shared memory segments, which were not fully handled before. We have implemented our approach in Rfblare, available at `https://blare-ids.org/rfblare`.

References

1. Cristiá, M., Mata, P.E.: Runtime enforcement of noninterference by duplicating processes and their memories. In: Workshop de Seguridad Informática WSEGI. vol. 2009 (2009)
2. Denning, D.E.: A Lattice Model of Secure Information Flow. Communications of the ACM 19(5), 236–243 (May 1976)
3. Georget, L., Jaume, M., Piolle, G., Tronel, F., Viet Triem Tong, V.: Verifying the Reliability of Operating System-Level Information Flow Control Systems in Linux. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. IEEE, Buenos Aires, Argentina (May 2017)
4. Hauser, C.: Détection d'intrusion dans les systèmes distribués par propagation de teinte au niveau noyau. Ph.D. thesis, University of Rennes 1, France (Jun 2013)
5. Jaume, M., Andriatsimandefitra, R., Viet Triem Tong, V., Mé, L.: Secure states versus Secure executions: From access control to flow control. In: International Conference on Information Systems Security. Springer (Dec 2013)
6. Krohn, M., Tromer, E.: Noninterference for a Practical DIFC-Based Operating System. In: IEEE Symposium on Security and Privacy. pp. 61–76. IEEE Computer Society, Washington, DC, USA (2009)
7. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: ACM SIGOPS symposium on Operating systems principles. pp. 321–334. ACM, Stevenson, WA, USA (Oct 2007)
8. Nadkarni, A., Andow, B., Enck, W., Jha, S.: Practical DIFC Enforcement on Android. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 1119–1136. USENIX Association, Austin, TX (Aug 2016)
9. Porter, D.E., Bond, M.D., Roy, I., Mckinley, K.S., Witchel, E.: Practical Fine-Grained Information Flow Control Using Laminar. ACM Transactions on Programming Languages and Systems 37(1), 1–51 (Nov 2014)
10. Roy, I., Porter, D.: Laminar (Aug 2014), `https://sourceforge.net/p/jikesrvm/research-archive/26`
11. Roy, I., Porter, D.E., Bond, M.D., McKinley, K.S., Witchel, E.: Laminar: Practical Fine-grained Decentralized Information Flow Control. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 63–74. ACM, Dublin, Ireland (Jun 2009)
12. The Coq Development Team: The Coq Proof Assistant Reference Manual. Tech. rep., Inria (Dec 2016)
13. VanDeBogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., Mazières, D.: Labels and Event Processes in the Asbestos Operating System. ACM Transactions on Computer Systems 25(4) (Dec 2007)
14. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux Security Modules: General Security Support for the Linux Kernel. In: USENIX Security Symposium. pp. 17–31. USENIX Association, San Francisco, CA, USA (2002)
15. Zimmermann, J., Mé, L., Bidan, C.: Experimenting with a Policy-Based HIDS Based on an Information Flow Control Model. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) (Dec 2003)