2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)

Verifying the Reliability of Operating System-Level Information Flow Control Systems in Linux

Laurent Georget* Mathieu Jaume† Guillaume Piolle* Frédéric Tronel* Valérie Viet Triem Tong*
*EPC CIDRE SUPELEC/INRIA/CNRS/University of Rennes 1 Rennes, France
†Sorbonne Universités, UPMC, CNRS, LIP6 UMR 7606 Paris, France
Email: laurent.georget@irisa.fr

***Abstract*—Information Flow Control at Operating System (OS) level features interesting properties and have been an active topic of research for years. However, no implementation can work reliably if there does not exist a way to correctly and precisely track all information flows occurring in the system. The existing implementations for Linux are based on the Linux Security Modules (LSM) framework which implements *hooks* at specific points in code where any security mechanism may interpose a security decision in the execution. However, previous works on the verification of LSM only addressed access control and no work has raised the question of the reliability of information flow control systems built on LSM. In this work, we present a compiler-assisted and reproducible static analysis on the Linux kernel to verify that the LSM hooks are correctly placed with respect to operations generating information flows so that LSM-based information flow monitors can properly track all information flows. Our results highlight flaws in LSM that we propose to solve, thus improving the suitability of this framework for the implementation of information flow monitors.**

***Keywords*-Information flow control; LSM; Static analysis**

I. Introduction

Information flow tracking is a security mechanism designed to monitor how sensitive information spreads in a system. Using this knowledge, information flow control can be applied to put limits on the dissemination of a piece of sensitive data once it is out of its *container of information* and let high level policies such as "my banking information shall not be sent to the Internet" or "information from accouting files from 2014 and 2015 shall not be mixed" be enforced easily. Another very active application domain of the information flow control (IFC) is the analysis of malwares [1], [2], [3] because, as obfuscated as its binary code may be, the information flows performed by a program always reveal its behavior.

We define an explicit information flow as the copy, usually partial, of the content of one container of information to another. The scope of this paper is the IFC at the Operating System (OS) level. For an OS, a container of information can be an active entity executing code (a process), a storage place (a file) or a channel of communication between active entities (a network socket, a shared memory segment, a pipe). For example, a process writing the content of a buffer in memory to a file generates an information flow from the process to the file. Information flow tracking consists in attaching a piece of meta-information on each container, called a *label*, that remembers where the information it contains comes from. Each time there is an information flow in the system, this should make the information flow tracker update the destination label according to the source label. Although mandatory access control models can also enforce information flow policies, we focus here on IFC performed with information flow *tracking*.

The flows are not usually performed directly by the processes but by the OS privileged software, the kernel, for security reasons. To write a file, or to start a new process, for example, a process has to ask the kernel to perform the desired task on its behalf by a *system call*, an entry point function implemented in the kernel. The kernel can deny any such request, if it lacks necessary resources or if the process lacks the required clearance. Therefore, the system call barrier is an ideal candidate to draw a clear boundary between containers of information and to define the granularity of the IFC. For instance, as all threads of execution in a single process share a single memory space, and can freely exchange information and work on the same data without requiring intervention from the kernel, we ought to consider the entire process, including all its threads, as a single container of information. We restrict ourselves to *explicit* information flows such as information storing or inter-process communications through means designed for this purpose. We thus exclude covert channels and information leakage due to some event or operation *not* occurring in the system.

Several approaches have been proposed to implement information flow tracking in Linux-based OSes. For instance, Blare [4], Laminar [5], and Weir [6] implement information flow tracking inside the Linux kernel. They do so thanks to the Linux Security Modules framework (LSM) [7], [8]. This framework was designed to allow any security mechanism be implemented efficiently and easily in the Linux kernel. Before security-sensitive operations (such as writing to a file, creating a new process, etc.), the kernel hands the execution out to the installed security module to let it take a security decision. The points in the code where security modules may interpose itself in the execution are called *hooks*. Although the formal validation of the position of the hooks has already been explored in details [9], [10], [11], [12], these works have not considered the information flow tracking use case. The main difference is that access control only needs to deal with granting or denying access to an object to a subject based on

978-1-5386-0422-9/17 $31.00 © 2017 IEEE
DOI 10.1109/FormaliSE.2017.1

a predetermined and mostly static policy and classification of objects and subjects according to their sensitivity and need-to-know. Information flow control takes into account the past interactions of a subject or object before taking a security decision. For example, it is possible for a top-secret subject to write into a public file only if said subject has not read secret information beforehand. The analysis of our results in Section V gives more insight on why a correct placement of hooks for the access control may not be correct for information flow monitoring purpose. To the best of our knowledge, our work is the first to formally assess the correctness of LSM hooks positioning for information flow tracking purposes.

Our work aims at verifying a necessary condition on the placement of hooks: that any execution path that generates an information flow goes through a LSM hook before the flow is performed. This is a necessary condition for the implementation of correct information flow trackers. Our main contributions are: (1) a formal model of the Linux code, produced by the GCC compiler itself, (2) a reproducible static analysis that for each system call, and each execution path reaching a point where an information flow is done, verifies that the path is either impossible or covered by a LSM hook (3) a study of the flaws in the design of LSM exhibited by our results and the corresponding patch to improve LSM's suitability for the implementation of information flow trackers. As a side note, we also make a case for compiler-assisted static analyses. Indeed, we are not able to define entirely the source language (a dialect of C specific to the compiler which is used, including parts written in assembly) but we can make working hypotheses on it and rely on the compiler to provide us with the knowledge that is hard to statically correctly decide such as the aliasing information (the C language supports arbitrary pointer arithmetic). Due to space restrictions, we do not expose all the formalism but details and proofs are available on the project website at https://kayrebt.gforge.inria.fr.

After giving in section II some background about our research, we present our analysis in the remaining sections. In Section III, we describe the formal model we use to represent the system calls. We want to formally verify that all execution paths which generate information flows but which do not contain a LSM hook are impossible paths. To do so, we equip the formal model with a semantics and design a static analysis we prove sound in Section IV. In section V, we present the GCC plugin we implemented to run our analysis and the results of our experimentations. Along with these results, we propose a revision of LSM with some hooks added, and some existing hooks relocated, to better adapt it to the control flow purpose. Finally, we discuss related work in section VI and conclude in Section VII with the further work we envision.

II. Background and Objectives

A. Linux Security Modules

Since version 2.6, Linux supports a framework to implement security extensions for the kernel called Linux Security Modules (LSM) [7]. This framework provides a set of *hooks* strategically placed in the kernel code associated with fields in internal data structures for exclusive use by security extensions. The hooks are functions which can be used by security extensions to (1), allocate, free, and maintain the security state of various internal data structures having a dedicated security field, and (2), implement security checks at specific points of execution, based on the security state and a policy. Security modules have a chance to apply security restrictions anywhere a hook is present, but only at these places. LSM's original design is the access control and this has dictated the placement of hooks in the code. It is thus necessary to verify the correctness of this placement for the purpose of information flow tracking to ensure that information flow trackers such as Blare, Laminar and Weir can operate properly.

B. Complete Mediation Property

Our main goal is to verify the property we call "Complete Mediation". For any execution path in the kernel starting with a system call and leading to an information flow, there is at least one LSM hook in this path which is reached before the flow is performed. The rationale here is that if a path generating a flow but not going through any LSM hook exists, then a malicious program can exploit it to perform illegal information flows without triggering any alarm because the information flow monitor can only react when one of the hooks is reached. Identifying all the paths that lead to information flows requires solving two common problems in static analysis. First of all, the number of execution paths is infinite because of loops in the code so we must identify a finite subset on which running the analysis is sufficient to draw a conclusion for all paths. Secondly, many execution paths that appear in the control flow graph of a program cannot actually be taken. These paths are often called *infeasible paths* in the litterature [13]. It is important to identify them, otherwise we might declare that the information flow monitor does not correctly detect some possible flows while in fact, there is no way to perform them.

For our analysis, we consider the list of system calls presented in Table I, which covers all the system calls monitored by either Blare, Laminar or Weir. Flow control requires knowing precisely when an information flow starts and when it stops. Otherwise, it is unable to maintain a correct and precise representation of all flows currently taking place in the system at any given time. This can result in either (1) wrong security decisions, compromising the confidentiality or integrity of sensitive data, or (2) overly conservative ones, making the system unusable. Access control on the other hand typically requires monitoring only the beginning of the flow (it would be inconsistent to authorize a process to gain access to a container but to forbid it to lose that access). Since LSM was designed with access control in mind, some hooks are missing to perform information flow tracking. Section V explore these points more in detail.

III. Modeling the System Calls Causing Flows

A. Control Flow Graphs

The analysis we propose relies on the C compiler from the Gnu Compilers Collection [14], used to compile the Linux

kernel. The model we designed to represent system calls and their execution paths does not describe directly the C source code but instead an internal representation created by GCC in a language called GIMPLE [15]. Each system call is represented by a control flow graph (CFG), where paths, as defined by the classical graph theory, model execution paths in the program [16]. Our analysis examines one system call at a time. To produce a single graph for an entire system call and all the functions it calls, we force the inlining of the latter into the former. This does not change the semantics of the code but reduces our analysis to the intra-procedural case. In these CFGs, we mark two kinds of nodes: the nodes corresponding to LSM hooks and the nodes corresponding to operations generating flows. The former can be identified automatically and we identified manually the latter in the CFGs (cf. Section III-B).

In our CFGs, contrarily to most approaches, a node is not a basic block but a single GIMPLE instruction. Edges can bear guards, to indicate conditional jumps between instructions. Our analysis does not need to deal with all expressions and variables of the language. For example, we do not have to handle floating-point values because their use is explicitly prohibited in the Linux code. We chose not to handle variables representing structures or unions when they involve pointer arithmetic. Powerful static analysis are allowed by generic framework such as Blast [17] but dealing with complex types is not necessary for our specifics needs. We do not handle global or volatile variables either since they can have an arbitrary value at any point in the execution, regardless of what is done in the function. Ignoring some variables does not hinder the soundness of our approach: less impossible paths might be detected as such but we never declare as impossible a possible path. Our results show that the analysis is precise enough to conclude in all cases.

To handle the pointer variables, we need to partition variables along two different criteria: (1) the set of pointers $\mathcal{V}ars^{ptr}$, i.e. variables whose value is the address of another variable, versus the set of integer variables $\mathcal{V}ars^{\mathbb{Z}}$; and (2) the set of variables whose address is taken at one point in the program $\mathcal{V}ars^{mem}$ versus the variables that are not addressable (not *aliasable*) and do not participate in side effects $\mathcal{V}ars^{temp}$ (i.e. their value can only be set by direct assignment). The segregation between the variables is directly extracted from the compiler, which maintains precise typing information about all expressions in the code.

In addition to variables, several kinds of expressions have to be handled: the integer constants $\mathbb{Z}$, the pointers dereferenciations $\{*p \mid p \in \mathcal{V}ars^{ptr}\}$, the address-taking operations $\{\&v \mid v \in \mathcal{V}ars^{mem}\}$, and finally the expressions of unknown value. As a matter of fact, all expressions leading to computations fall into the latter category, as well as unanalyzed variables. Finally, our analysis relies on the *points-to* oracle maintained by GCC, which gives for any pointer the set of variables it can possibly points to.

A path in the CFG is said to be impossible when any execution that would follow it would enter in an impossible state. For example, a path including two conditional branchings with incompatible conditions would require a boolean expression to be both true and false at the same time.

B. Rationale and Discussion of our Approach

Using the compiler itself to extract the knowledge we need from the kernel has several advantages. First of all, we have the guarantee that the analysis is bound to the exact code that is compiled in. Due to conditional compilation, the code of a given function can vary greatly depending on the compilation options. Moreover, it is important to note that GCC's extensions to the C language are explicitly permitted. Relying on GCC is therefore not only useful but practically mandatory to correctly capture the true semantics of the code. Finally, even if the representation we extract during the compilation process and use as a model of the code does not reflect what the developpers *intended to* mean, it is anyway what the code *does mean* when executed.

Another decision we made was declaring where information flows are actually performed inside the system calls. This is not trivial because the low-level data structures manipulated in the kernel code sometimes only relates loosely to the abstractions we call "files" or "sockets". For instance, writing to a file on disk actually means writing to a cache in memory, which is periodically flushed to the disk by a dedicated kernel process. It is natural to consider that when the writing process returns from `write` system call, the information flow is done because any process reading the file would find the information just written there but in case of a power outage for example, the information could be lost if it had not been flushed to disk yet. We used several heuristics to decide objectively that the flow was performed at some point in the code: the fact that all checks were passed, the fact that all locks on structures were taken, and finally the fact that the operation was "committed" by the kernel by a log message for example.

IV. Static Analysis on Paths

Our objective is, for any CFG representing a system call generating an information flow, to enumerate all the paths of this CFG reaching the point where the information flow is done and for each one of these paths, to verify that it goes through a LSM hook or that it is impossible. If there exists a path which is possible but not covered by a LSM hook, then this is a flaw in the placement of hooks. In this section, we describe the main lines of the static analysis and the main results we have proved onto it.

A. Verifying the Complete Mediation

We consider the set $\mathcal{P}aths$ of all paths in a CFG. Among this paths, we distinguish two particular subsets: (1) the set $\mathcal{P}aths_{flows}$ of paths starting at the initial node of the CFG and ending at one node generating an information flow; and (2) the set $\mathcal{P}aths_{LSM}$ of paths having a node corresponding to a LSM hook. The placement of the hooks would be obviously correct if we could prove that $\mathcal{P}aths_{flows} \subseteq \mathcal{P}aths_{LSM}$. However, as

we will see, this is not the case. $\mathbf{P_f} = \mathcal{P}aths_{flows} \setminus \mathcal{P}aths_{LSM}$ represents the set of paths that may be problematic.

As explained earlier, some paths in $\mathcal{P}aths_{flows}$ are actually impossible, and therefore even if there are no LSM hooks in them, they are not actually problematic. Recall that an impossible path is a path that does not correspond to a possible execution. The objective of our analysis is thus to verify that $\mathbf{P_f} \subseteq \mathbf{I}$ where $\mathbf{I} \subseteq \mathcal{P}aths$ is the set of impossible paths in the CFG. However, due to the presence of loops, the set $\mathbf{P_f}$ is infinite and we cannot run our analysis on all paths in $\mathbf{P_f}$. To tackle this problem, we define an equivalence relation on paths in $\mathbf{P_f}$: two paths are equivalent if they are identical up to the cycles they contain. Our analysis run only on acyclic paths (of which there is a finite number) and we prove that our analysis is conservative, i.e. that it will not report a path as impossible if at least one equivalent path is possible.

Of course, in any CFG there is a finite number of acyclic paths, so it is possible (and sufficient) to verify them all. We state the complete mediation property:

Property 1 (Complete Mediation):
Complete mediation holds iff: $\mathbf{P_f} \subseteq \mathbf{I}$, i.e. all the execution paths that perform an information flow and are not controlled by the information flow monitor since they do not contain a LSM hook are impossible according to the static analysis.

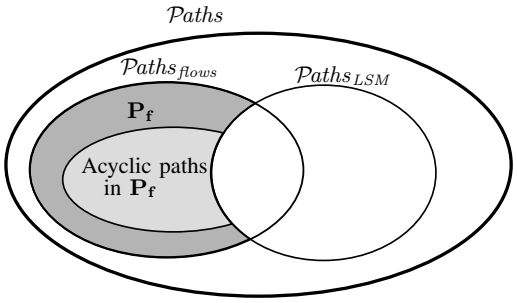

Figure 1. Sets involved in the anaysis

B. Configurations and Satisfiability

The analysis of a path requires maintaining an abstraction of the possible state of the execution at each node. This state, which we call an *abstract configuration*, is twofold: the path configuration contains information about the path followed so far by the execution in the graph and the variables configuration contains constraints about the possible values of variables. Our analysis is said to be *path-sensitive* because it leverages the knowledge of the execution path taken. Actually, due to the properties of the graphs and the needs of our analysis, it is only necessary to remember the edge taken to reach the last node with strictly more than one predecessor node (i.e. the node at the beginning of the current basic block).

A variables configuration written (C, P) is made up of a *set of constraints* C on the integer variables and a mapping P for the pointers. C is a set of tuples of the form (x, R, y) where x is a variable, $R \in \{=, \neq, <, >, \leq, \geq\}$ a relation operator, and y either a variable or an integer constant. Such a constraint (x, R, y) expresses the fact that the predicate $\bar{x}\ R\ \bar{y}$, where $\bar{v}$ stands for the value of v, holds. The pointer mapping P records the value of pointers, if known. We have $P : \mathcal{V}ars^{ptr} \rightarrow \mathcal{V}ars^{mem} \cup \{\top\}$. With $p \in \mathcal{V}ars^{ptr}$, if $P(p) = v \in \mathcal{V}ars^{mem}$ then we know that p points to v, otherwise, we have no more information than the points-to oracle. Indeed, since our analysis is path-sensitive, it has sometimes more information about the value of a pointer than the path-insensitive points-to analysis made by the compiler. The configurations are updated by following the execution path: assignments add constraints on variables or, on the contrary, remove some. For example, when a variable is assigned the return value of a function, the analysis remove all constraints about the variable since the return value of a function is statically undecidable in the general case. An edge always adds as a constraint the condition it bears, if any.

Our problem of identifying impossible paths is thus reduced to a problem of satisfiability: if there does not exist a valuation for the variables that satisfies all the constraints simultaneously, then we consider that we have reached an impossible state of execution. This corresponds to incompatible constraints and means that the followed path does not correspond to a possible execution. If the set of constraints is satisfiable, or at least if the analysis cannot prove it is unsatisfiable, then we declare the path possible.

C. Concrete Configurations

In order to prove the soundness of the static analysis, we have to describe the concrete execution. We define the *concrete configuration* as a valuation of the variables of the program, at a given point in its execution. First of all, we consider the memory as a set of memory cells, each one holding a single value, which may be either an integer or the identifier of another memory cell (an "address"). A concrete configuration has three components: (1) a mapping from memory cells to values, (2) a mapping from non-addressable variables to values, and (3) a mapping from addressable variables to memory cells. The first mapping is the memory state of the program. The second one is used to deal with the particular case of non-addressable variables which might not live in memory. Although they appear in the CFG built by the compiler, they have a single value during their lifespan and are not susceptible to be modified via an indirect assignment such as a pointer for example, so it is not necessary to account for them in the memory state. In fact, although they appear in the CFG built by the compiler, they might very well be optimized out in the resulting executable program.

D. Concrete and Abstract Semantics

We aim at proving the soundness of our static analysis: if a path is declared as impossible by the static analysis, then it is indeed impossible, although the analysis might not detect all impossible paths. To prove this property, we suppose the existence of a concrete semantics of the CFG, the one computed by the compiler. Since the C language, and the compiler's internal representations, including the CFG, have

only informal semantics, we only make minimal, common hypotheses on the concrete semantics, necessary to our analysis, without defining it entirely. This way, we eliminate the risk of diverging from the compiler's actual semantics. The concrete semantics is a transition relation $\rightarrow \subseteq \Theta \times \mathcal{V} \times \mathcal{C} \times \Theta$ where Θ is the set of concrete configurations, $\mathcal{V}$ the set of nodes and $\mathcal{C} = (\mathcal{V}ars \times \{=, \neq, <, >, \leq, \geq\} \times (\mathcal{V}ars \cup \mathbb{Z})) \cup \{\texttt{true}\}$ is the set of constraints on edges. `true` is a special constraint which is always satisfied (to model unconditional edges).

We then proceed to completely define an abstract semantics. The abstract configuration is a transition relation $\twoheadrightarrow \subseteq \mathcal{K} \times \mathcal{V} \times \mathcal{C} \times \mathcal{K}$ where $\mathcal{K}$ is the set of abstract configurations. The entire definition can be found in the extended appendix of the paper, available on the project's website. We extend naturally these relations over paths.

To express the soundness of our analysis, we define a satisfiability relation between concrete and abstract configurations. We say that a concrete configuration satisfies an abstract one if the valuation of the variables given by the concrete configuration satisfies all the constraints in the abstract configuration. Noting θ a concrete configuration, and k an abstract one, we write $\theta \Vdash k$ when θ satisfies k and $\Vdash k$ the fact that k is satisfiable, i.e. there exists some θ such that $\theta \Vdash k$.

The soundness of our analysis is stated as follows.

Proposition 1 (Soundness):
For each path p in a CFG, for all concrete configurations θ_1 and θ_2, and all abstract configurations k_1 and k_2 such that $\theta_1 \rightarrow_p^* \theta_2$ and $k_1 \twoheadrightarrow_p^* k_2$, we have $\theta_1 \Vdash k_1 \implies \theta_2 \Vdash k_2$.

We define the set of executable paths as the set of all paths p such that there exists two concrete configuration θ_1 and θ_2 such that $\theta_1 \rightarrow_p \theta_2$. Remember that a transition may not exist from one configuration to another if, along the path p, there is an edge bearing a condition which is not verified by the memory state. The set of impossible paths is defined as the set of paths p for which given any two abstract configurations k_1 and k_2, if $k_1 \twoheadrightarrow_p^* k_2$, then $\nVdash k_2$. In other words, there is no satisfiable configuration that can result from the analysis of path p. The following proposition is an immediate consequence of the previous one.

Proposition 2 (Executable vs. impossible paths): The sets of executable paths and impossible paths are disjoint.

E. Handling Loops

The static analysis, as presented so far, handles all paths of finite length. However, in a CFG, there are an infinity of such paths, because of the presence of loops. Fortunately, loops have a special syntax in CFGs. Specifically, loops must start with one join node with exactly two predecessors: one before the loop, and one inside. Loops are also disambiguated by the compiler: a unique node cannot be the begining of two different loops. Furthermore, for any node, we can use the compiler as an oracle to tell what is the most-outer loop it is part of, if any, and whether it is the beginning of a loop. We make the hypothesis that the number of iterations of each loop is unbounded but finite, since it would actually be a bug for a system call not to terminate. We deal with loops in our analysis by computing a loop abstractor, i.e. a configuration such that if it is unsatisfiable, then all abstract configurations that could possibly result from the analysis of this loop are unsatisfiable. In other words, we abstract in a single equivalence class all the paths composed of a finite, albeit arbitrarily large, number of iterations of the loop. Our analysis therefore trades off precision for termination by reducing the analysis of any loop to the computation of its abstractor.

V. Implementation and Results

The static analysis is run during the compilation itself. To this effect, we developped a plugin for GCC version 4.8. This plugin is inserted into the compilation process, when the code is in GIMPLE form. We chose to place the plugin as late as possible in the compilation process, just before GCC abandons the graph intermediate representation, in order to benefit the most from optimizations and to work on a code as simple as possible. This allows us to do the assumptions we presented in the previous sections, such as the particular properties of loops. The representation we dump is broken down to very elementary pieces. The code we handle is in three-addresses mode, which has the effect that all complex boolean conditions that could exist in the original code base are already broken down in the appropriate number of binary decision nodes and branchings. We also leverage as much as possible the compiler. For example, we rely on it to identify the loops in the CFG, to know the precise typing information of variables, and to run the points-to analysis on pointers. The latter is already implemented because it is used for several optimizations passes by GCC. Finally, the careful use of inlining and the fact that GCC applies some optimizations on the code actually limits the path explosion. Our tools, including the plugin for the static analysis and also a plugin able to dump CFGs, are available at http://kayrebt.gforge.inria.fr.

Prior to running the analysis, we place a special annotation on places in the code of system calls where information flows are performed. When the analysis is run on a system call, for each annotated places, a subgraph of the CFG built by GCC is considered: the subgraph of paths starting from the node representing the entry of the system call and going to the information flow node that do not pass through any LSM hook. These paths are the set $\mathbf{P_f}$. For each path, we start with an empty abstract configuration and we update the configuration as we go along the path as per the transition rules of the abstract semantics. The satisfiability of the abstract configuration is tested by Yices [18], a SAT-solver equipped with a decidable subset of the classic theory of integers. If the constraint solver declares the set of constraints unsatisfiable then we declare the path impossible.

We did our analysis on the Linux kernel version 4.3, released on November 2nd 2015, compiled with the default configuration options for architecture x86_64. However, the same analysis can be reproduced with the same tools on any version, any platform. Most of the code we analyze is part of high-level submodules, which are not likely to change

greatly from one version to the next, and are not architecture-dependent. Nonetheless, the list of available system calls depends on the configuration options, therefore, the options available for performing information flows can vary from a Linux system to another. The results of our analysis are presented in Table I, where system calls are grouped by the LSM hooks they share. In a majority of cases, either no paths evading the hooks are found or they are all impossible. In these cases, LSM hooks are placed correctly to detect the information flows generated by these system calls. In some other cases, though, some system calls we have identified as generating flows have no LSM hooks at all.

a) System calls `vmsplice`*,* `splice`*, and* `tee`*:* These system calls take advantage of the implementation of the pipes as memory buffers to perform efficient copies to or from pipes. No hook is triggered when the flow occurs between two pipes.

b) `mq_timedreceive`*,* `mq_timedsend`*:* These system calls are used to, respectively, receive and send a message through a POSIX message queue. They behave the same as `msgrcv` and `msgsnd` for System V message queues, which do have hooks. We got some elements of explanations on the LSM kernel development mailing-list from SELinux developper Stephen Smalley [19]. First, these system calls are more recent than LSM so they were not included in the original LSM design. Secondly, the difference between the two message queues APIs is that the POSIX one is based on the virtual filesystem. For the purpose of access control, it already benefits from the security hooks in the `open` system call. However, and this also applies to `splice`, in the system calls that do the actual information flow, there are no hooks because pipes and message queues are not thought as being subject to the same problem of policy change and access control revalidation as regular files, and therefore it feels less necessary to place a hook before each individual "read" and "write" operation as for files. These considerations are valid only when considering access control. Flow tracking requires monitoring each individual information flow.

c) `process_vm_readv`*,* `process_vm_writev`*:* In these system calls, two paths are possible. However, they are designed as such and correspond to flows from a process to itself (possibly from one thread to another). In our model, a process as a whole is a container of information because flows between threads are possible without system calls anyway.

d) `recvmmsg`*,* `sendmmsg`*:* The case of these system calls illustrates why a framework built for access control might be insufficient for flow control. The system call `recvmmsg` allows a process to receive multiple messages from a socket with a single system call. A process may use it to communicate with another one. A LSM hook is present in this system call, so that the communication is mediated. However, the hook is positioned such that the security function is called only once, before the first message is received. A malicious process could exploit this fact to send sensitive information acquired (via another thread) between the first and the subsequent messages. This is a problem relevant only to flow control and not to access control. With access control, the dynamic content of a container of information has no influence over the security decision, therefore it is sufficient to do the check only once per system call.

VI. Related Work

To the best of our knowledge, no previous work has aimed at assessing the adequacy of LSM for information flow tracking. However, several works have focused on the correctness of the placement of the LSM hooks for its original purpose: access control. Zhang et al. performed a static analysis with CQUAL [20] to check if all accesses to internal data structures are correctly intercepted by the appropriate LSM hooks. This is different from our approach because we consider more abstract structures, containers of information, whose link with the real data structures of the kernel is not always direct. This work was extended [9] with a consistency analysis on the relationships between the security-sensitive operations performed by the kernel and LSM hooks. In this work, Zhang and Jaeger identified all the paths that can lead to an access to a member of an internal data structure at a granularity finer from that of system calls. For example, with their analysis, they have been able to detect that the `f_owner.pid` member of `struct file` can be modified via an operation which, as a side-effect, resets this field to 0 without calling the hook designed to protect this field before. Again, this approach is too low-level for us because we deal with more abstract objects. Nevertheless, our future work will adapt from this approach the notion of *consistency*. In this current work, we have only checked that any flow is monitored by *some* hook. Later works by Ganapathy, Jaeger, and Jha [10] and Muthukumaran, Jaeger, and Ganapathy [12] focused on automatic placement of hooks. This approach is promising but not practical in our case since we do not use the LSM framework for its primary purpose but we want to modify it as little as possible nevertheless. Bringing IFC in Linux through LSM is an active topic and adding hooks or moving some of them has been necessary for *Blare* and *Laminar* [4], [21], even if the placement of LSM hooks have already been carefully analyzed. This encourages us to verify the correctness of hooks placement for information flow tracking purposes.

VII. Perspectives and Conclusion

In this paper, we have presented an approach to verify the suitability of LSM for the implementation of information flow trackers. We have shown that the compilation process can be leveraged to build a formal model of the source code to run single-purpose static analysis. Indeed, the compiler already needs to build a control flow graph, and maintains precious knowledge such as a points-to analysis to compile and optimize the code, and all this can be reused for more static analysis on the code. The clearest advantage of this method is the prevention of any risk of making errors in the semantics of the code. We designed and proved a reproducible static analysis on the position of the LSM hooks to track information flows. This analysis is simple, specifically adapted for its purpose and fast. We then examined the results and

Table I
PRESENCE OF PATHS ESCAPING ALL LSM HOOKS IN THE SYSTEM CALLS

read pread64
readv preadv writev pwritev
write pwrite64
sendfile sendfile64
shmat
mmap

(a) System calls in which all paths are mediated by at least one LSM hook

msgrcv
msgsnd
clone fork vfork
execve execveat
send sendto sendmsg
recv recvfrom recvmsg

shmdt
vmsplice
tee
munmap
mq_timedsend
mq_timedreceive

(b) System calls producing information flows but having no LSM hooks

process_vm_readv process_vm_writev	two cases of possible paths: a thread reading another thread's memory in the same process and a thread reading its own memory
splice	no hooks for the pipe to pipe information flow no paths for the file to pipe information flow no paths for the pipe to file information flow
sendmmsg (same hooks as send)	the LSM hook is hit only once, for the first message, for the other messages sent with the same system call to the same destination, the hook is not called again
recvmmsg (same hooks as recv)	the LSM hook is hit only once, for the first message, for the other messages read with the same system call, the hook is not called again
ptrace	no paths for the attachment, but no hooks for the detachment

(c) System calls where not all paths are mediated

concluded that it was possible, after some modifications, to use LSM to track individual information flows. Finally, we have written a patch for the Linux kernel, version 4.3, to place the LSM hooks where we suggest. We validated the new placement by redoing the static analysis. The patch can be found at http://kayrebt.gforge.inria.fr/pathexaminer.html. This work validates the approach of developping information flow trackers on top of LSM and provides a strong basis for the formal assessment of their correctness.

We have shown that the LSM frameworks must be extended to cover all execution paths generating information flows on containers of information. This is a necessary condition to correctly implement information flow control but we plan also to consider further issues relative to the way the hooks are used. For example, when simultaneous information flows involving the same containers are considered, it is important that the tags are propagated in the same order than the flows are performed. These problems are already under investigation by the authors.

REFERENCES

[1] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. Viet Triem Tong, "GroddDroid: a gorilla for triggering malicious behaviors." IEEE, Oct. 2015.

[2] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *ACM Conference on Computer and Communications Security*. ACM, 2007.

[3] L. K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *USENIX Conference on Security Symposium*, ser. Security'12. USENIX Association, 2012.

[4] L. Georges, V. V. T. Tong, and L. Mé, "Blare Tools: A Policy-Based Intrusion Detection System Automatically Set by the Security Policy," in *International Symposium on Recent Advances in Intrusion Detection (RAID 2009)*, 2009.

[5] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical Fine-grained Decentralized Information Flow Control," SIGPLAN Not., vol. 44, no. 6, 2009.

[6] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC enforcement on Android," in *USENIX Security Symposium*, 2016.

[7] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *USENIX Security Symposium*. USENIX Association, 2002.

[8] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, 2001.

[9] T. Jaeger, A. Edwards, and X. Zhang, "Consistency analysis of authorization hook placement in the Linux security modules framework," ACM Trans. Inf. Syst. Secur., vol. 7, no. 2, 2004.

[10] V. Ganapathy, T. Jaeger, and S. Jha, "Automatic Placement of Authorization Hooks in the Linux Security Modules Framework," in *ACM Conference on Computer and Communications Security*. ACM, 2005.

[11] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka, "Verifying information flow goals in security-enhanced Linux," J. Comput. Secur., vol. 13, no. 1, 2005.

[12] D. Muthukumaran, T. Jaeger, and V. Ganapathy, "Leveraging "choice" to automate authorization hook placement." ACM Conference on Computer and Communications Security, 2012.

[13] R. Bodík, R. Gupta, and M. L. Soffa, "Refining Data Flow Information Using Infeasible Paths," SIGSOFT Software Engineering Notes, vol. 22, no. 6, Nov. 1997.

[14] R. M. Stallman and the GCC developer community, "Using the GNU Compiler Collection (GCC)," Tech. Rep., 2013. [Online]. https://gcc.gnu.org/onlinedocs/gcc-4.8.4/gcc/

[15] J. Merrill, "GENERIC and GIMPLE: A new tree representation for entire functions," in *GCC Developers Summit*, 2003.

[16] F. E. Allen, "Control Flow Analysis," in *Symposium on Compiler Optimization*. ACM, 1970.

[17] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," International Journal on Software Tools for Technology Transfer, vol. 9, no. 5, 2007.

[18] B. Dutertre and L. de Moura, "The Yices SMT solver," SRI International, Tech. Rep., 2006.

[19] S. Smalley, "[PATCH 0/2] Add missing LSM hooks in mq_timed{send,receive} and splice, Email on the Linux Security Modules development mailing list," 2016. [Online]. http://thread.gmane.org/gmane.linux.kernel.lsm/28737

[20] X. Zhang, A. Edwards, and T. Jaeger, "Using CQUAL for Static Analysis of Authorization Hook Placement," in *USENIX Security Symposium*. USENIX Association, 2002.

[21] D. E. Porter, M. D. Bond, I. Roy, K. S. Mckinley, and E. Witchel, "Practical Fine-Grained Information Flow Control Using Laminar," ACM Transactions on Programming Languages and Systems, vol. 37, no. 1, Nov. 2014.