Linux security modules and whole-system provenance capture

Aarti Kashyap
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
kaarti.sr@gmail.com

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it. - Frank Herbert

ABSTRACT

Data provenance describes how data came to be in its present form. It includes data sources and the transformations that have been applied to them. There have been several different OS provenance capture tools in the past. However, only CamFlow and its predecessor Linux Provenance Modules use Linux Security Modules that is a framework that allows the Linux kernel to support a variety of computer security models while avoiding favouritism toward any single security implementation. This paper examines the relationship between LSM and Camflow. The two key research questions that are addressed: 1) Does the last stable version of Linux kernel(v4.20) capture all information flows? 2) Given the results of (1) and the data captured by a whole-system provenance capture mechanism utilizing LSM hooks, can we prove that an intrusion will be reflected as an anomaly in the provenance graphs. The part 1 of our work focuses on ensuring that every information flow in v4.2 goes through a LSM hook. This is an important question that helps in validating the use of kernel provenance for intrusion detection. We found two system calls which have flows that don't pass through the LSM hooks.

I. INTRODUCTION

A provenance-aware system automatically gathers and reports metadata that describes the history of each object being processed on the system. This allows users to track, and understand, how a piece of data came to exist in its current state. The application of provenance is presently of enormous interest in a variety of disparate communities including scientific data processing, databases, software development, and storage [33, 34]. Provenance has also been demonstrated to be of great value to security by identifying malicious activity in data centers [35, 36, 37], improving Mandatory Access Control (MAC) labels [38,39,40], and assuring regulatory compliance [3].

Data provenance can help detect such intrusions in the kernel. It only provides with the capability to detect intrusions not prevent them. However, in order to make sure that all the intrusions are getting detected, we need a way to capture the complete data flow in the system which is why we chose to work with Camflow. Camflow is a practical implementation of whole-system provenance capture that can be easily maintained and deployed. They use Linux security modules and Netfilter hooks as the underlying framework to capture the data flows which provides us the ability to capture all the information flows. This ability to capture all information flows in a system is defined as whole-system provenance capture.

In this paper, we examine if the whole system provenance capture mechanism developed by Camflow can be utilized for intrusion detection. Camflow uses exisiting capture techniques provided by the functionalities of Linux operating systems. The work we have undertaken focuses on information security, and more specifically on the two security properties which should hold throughout the system.

1) Confidentiality: Only authorized users should have access to the information. In order to make sure that only authorized users have access to certain files in the Linux operating system, a general lightweight access control framework was developed. This was the Linux security modules(LSM) project(25,26). A number of existing acess control implementations including SELinux(27) were adapted to use LSMs.

2) Integrity The information stored in the system should not undergo any change. The crash of a PhD thesis in progress in word without any data recovery or the unauthorized creation of a user account in service by a hacker are some examples of intolerable corruptions.

In order to meet the needs of confidentiality and integrity, system administrators assign number of security policies. Hence, each user permissions and object of the system contains an information security level which limits the access. Object in this case is defined as the abstraction of the system which may contain information.

Security is a major concern since there is no permanent fix to detect intrusions. Its a race between attackers and defenders. An example to show that the kernel is still under threat Xioo

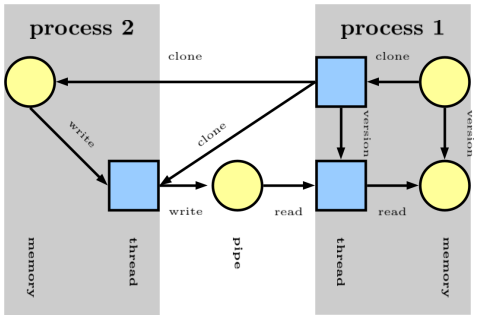

Fig. 1. Process 1 clones process 2. Process 2 writes to a pipe. Process 1 read from the same pipe

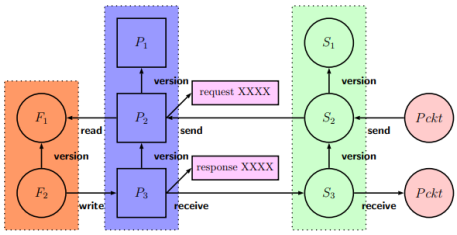

Fig. 2. Annotated provenance graph

et al.[22] showed that attackers are able to manipulate the running behaviours of operating systems without injecting any malicious code. This type of an attack is called as kernel data attack. With the power of tampering data, the attackers can stealthily subvert various kernel security mechanisms. These policies express security needs. Hence the first part of our work focuses on one of these mechanisms: monitoring information flow to ensure that the integrity property holds.

We first focus on a methodology proposed by Georget et al.[3] to verify if indeed every security related flow goes through an LSM hook. The methodology proposed by Georget et al. had been designed for linux kernel v4.3 to ensure that all security flows are passing through the hooks. We guarantee the same for v4.20 which is the last formal release. The tools developed by Georget et al.[3], however, can be utilized for performing static analysis with any kernel version. A second challenge in determining if the use of kernel provenance can help in intrusion detection, we prove that a security breach is reflected in the provenance graph it produces.

The key contributions of our work include: 1) ensuring if all information flows pass through the Linux security modules hooks. This work focuses on understanding if the placement of hooks is correct. 2) if Camflow uses the LSM and Netfilter hooks to capture the data, in case of intrusions the provenance graph is able to capture it.

II. Background and Objectives

To provide context for the rest of the paper, we first introduce a few concepts. We introduce the notion of whole-system provenance, provenance graphs, Linux security modules and complete mediation property.

A. Whole-system provenance capture

Data provenance was originally introduced to understand the origin of data in a database. [17, 18] According to the W3C [19] standards provenance is defined as a directed acyclic graph (DAG). The vertices in the DAG represent entities(data), activities(transformations of data) and agents(persons or organizations). The edges represent the relationships between these elements. Mapping the graph definition to our system, which is the OS level entities are kernel objects, such as messages, network packets, files etc., but also xattributes, inode attributes, exec call parameters, network addresses, etc. Activities are the tasks or processes carrying out manipulations on entities causing data flows. The agents are the persons or the organizations, who control the activities on different entities. These are the users and the groups at the OS level. the agents are users and groups. Fig 1 illustrates these concepts. In the example in Fig 1, process 1 clones process 2. Process 2 writes to a pipe and finally process 1 read from the same pipe.

Processes exchange information via system calls. Some system calls represent information exchange at some discrete point in time e.g, read, write; others may create shared states, e.g, nmap.

There have been multiple whole system provenance capture systems proposed in the past such as HiFi[11], PASS[10]. However, they had a couple of problems: 1) struggled to keep abreast with current OS releases 2) Did not have whole system provenance capture guarantees 3) generated too much data 4)imposed too much overhead. Learning from the lessons from the past the whole system provenance capture systems, Camflow was introduced which promised to resolve all the above mentioned issues. Camflow addressed the above mentioned shortcomings 1) by leveraging the latest kernel defining to achieve efficiency 2) using a self-contained, easily maintainable implementation relying on Linux Security Modules, Netfilter, and other existing kernel facilities.

B. Linux Security Modules

Since the kernel version 2.6, Linux added support for a framework to implement security extensions for the kernel called Linux Kernel Modules(LSM)(14). This framework provides a set of hooks strategically placed in the kernel code associated with felds 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. LSMs 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.

Since, Camflow uses LSMs and Net-Filters to capture the system provenance, the need to verify the correct placement of hooks for Camflow becomes necessary.

C. Complete Mediation Property

The first goal of our contribution is to verify the property called "Complete Mediation property". According to this property 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 the path which is reached before the flow is performed. It's important to verify this property. The reason is because if there exists a path generating a flow but not going through any LSM hooks, then there exists an opening for a malicious program to perform illegal actions without triggering any alarms. This is because information flow monitor can only react when one of the hooks is reached.

In order to identify all the paths which lead to information flows requires solving two common problems in static analysis. The first problem is that the number of execution paths is infinite because of loops and recursions in the code. In order to finitize the code a subset is selected which should be sufficient to draw a conclusion for all the paths. In other words constructing an abstraction of the code which can be mapped to the concrete code after the analysis is required. The second problem is that many execution paths that appear in the control flow graph (CFG) cannot actually be taken. These are called as the infeasible paths[20].

For our analysis we consider all the system calls in the kernel version 4.20. Flow control requires knowing precisely when an information flow starts and when it stops. If this information is not available, it is not possible to maintain a correct representation of all flows currently taking place in the system at any given time. Since LSM was designed with access control in mind, some hooks might be missing to perform information flow tracking in every kernel update that comes. However, using static analysis we can find if some hooks are missing to perform information flow tracking.

The purpose of information flow control is to monitor the way in which information is disseminated in the system once it is out of its original container. This is unlike access control which can only enforce rules on how whose containers are accessed. Several scientific and technical challenges exist in ensuring complete information flow. One of them being the large Linux kernel code base. Georget et al.[3] tackles this issue in his work.

The most common way to meet the security objectives is access control. In order to ensure this, each level of security permissions is associated with the read or modifications of objects assigned to this level. This makes sure that only authorized users can read or alter information when stored in the object marked at the correct level of security. However, the fragility of this approach lies in the fact that once the information is out of their original container, the policy can no longer protect the information. To fully protect the information, the policy must be transitive. If Alice has access to a file but Bob does not, then we should make sure that Alice does not have some means to communicate with Bob. If Alice passes the information, even accidentally, then that is the violation of the confidentiality system.

The information flow control responds to the above problem. By keeping a track of information movement that took place between objects in the history of the system, it is able to protect information even when outside of its original container. This is not limited as the communications in the case of access control. As per the above Alice and Bob example, Alice has the right to contact Bob, until the knowledge to which Bob does not have access to is communicated. If that is indeed the case, then the communication has to stop.

Hence, the initial Linux security module framework was built to support the access control mechanisms. We want to ensure that the framework is suitable enough to implement information flow mechanisms.

Camflow which utilizes LSM for the whole-system provenance capture. It collects the provenance data and constructs provenance graphs from the collected data. Now that we are aware that Georget's methodology ensures the placement of hooks such that complete information flow is possible, we show that the violations are reflected in the provenance graphs.

III. LSMs and LSM hooks

Though we have introduced what a Linux Security Module(LSM) means, we will discuss it in a little more detail. The Linux Security Module (LSM) framework provides a mechanism for various security checks to be hooked by new kernel extensions. The name module is a bit of a misnomer since these extensions are not actually loadable kernel modules. Instead, they are selectable at build-time via CONFIG_DEFAULT_SECURITY and can be overridden at boot-time via the "security=..." kernel command line argument, in the case where multiple LSMs were built into a given kernel.

A. LSMs and policies

The primary users of the LSM interface are Mandatory Access Control (MAC) extensions which provide a comprehensive security policy. Examples include SELinux, Smack, Tomoyo, and AppArmor. In addition to the larger MAC extensions, other extensions can be built using the LSM to provide specific changes to system operation when these tweaks are not available in the core functionality of Linux itself.

B. Types of LSMs

Without a specific LSM built into the kernel, the default LSM will be the Linux capabilities system. Most LSMs choose to extend the capabilities system, building their checks on top of the defined capability hooks.

The different types of Linux Security Modules are listed in Table 1. The entire list of the LSMs can be found by reading */sys/kernel/security/lsm*. The Table 1 reflects the order in which checks are made. The capability module is always first, followed by "minor" e.g. Yama) and then the one

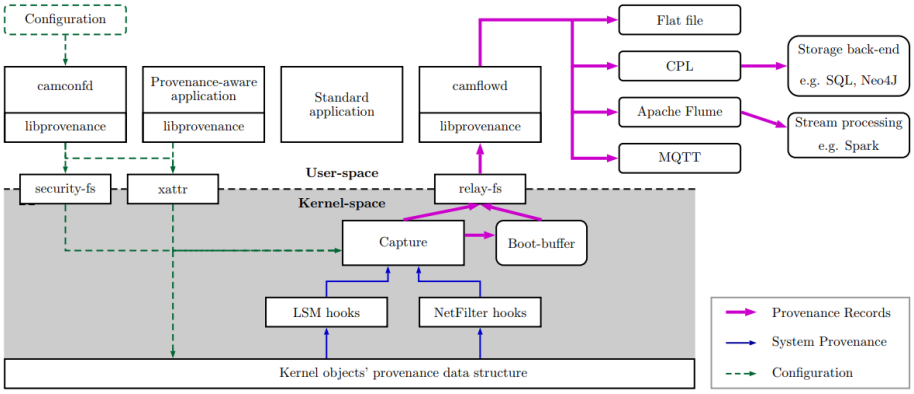

Fig. 3. Camflow architure

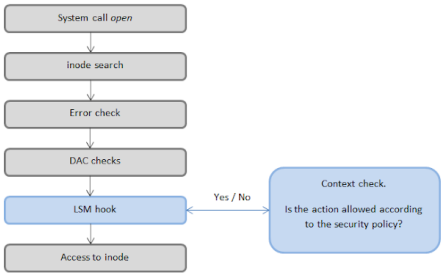

Fig. 4. LSM hook working for system call "open"

major module (e.g. SELinux) if there is one configured. This information will help us in understanding the next part of our research.

C. LSM Hooks

In order to explain how a LSM hook works, we show an example in Fig 4. We take system call open as an example. We can see in Fig 4 that just before the kernel addresses an internal object, a check function provided by LSM is called. Hence, LSM allows the modules to understand wheather subject S is allowed to perform an action OP over kernel's internal object OBJ.

TABLE I
LINUX SECURITY MODULES

Different LSMs
AppArmor
LoadPin
SELinux
Smack
TOMOYO
Yama

IV. MODELLING THE SYSTEM CALLS CAUSING FLOWS

Analysis of programs aims verification of certain properties expected of them. Conventionally, there are two categories of anlysis, dynamic analysis and static analysis. Static analysis checks the properties on the souce code of the program, according to the declared semantics of the programming language, without the need to compile the program into machine language. The analysis proposed relies on the C compiler from the Gnu Compilers Collection [23], used to compile the Linux kernel.

A. Control flow graphs

The analysis technique we use if has been proposed by Georget et al. () for a subset of the system calls. It's a four step methodology which relies on the C compiler from the Gnu Compilers Collection(21).

1) The model designed by Georget to represent system calls and their execution paths does not describe the C source code. They instead use an internal representation called GIMPLE[].
2) Each system call is represented by a control flow graph (CFG).
3) The paths in these graphs model the execution paths in the program as defined by the classical graph theory[].
4) The system calls are analysed one at a time.
5) Each system call contains multiple functions. These functions are inlined into the system calls to reduce the analysis to intra-procedural case.
6) Finally, in the CFGs, two kinds of nodes are marked: the nodes which correspond to the LSM hooks and the nodes which correspond to operation which generate the flows.

B. Constraints in modelling

In the CFGs which we construct , a node is not a basic block but a simple GIMPLE instruction. The analysis methodology does not deal with all expressions and variables of the language. Another reason for the same is that usage of floating-point values is explicitly prohibited in the Linux code. Variables representing structures or unions are also not handled when they involve pointer arithmetic. Global or volatile variables are also not handled, since they can have an arbitrary value at any point in the execution. 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. 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 branching with incompatible conditions would require a Boolean expression to be both true and false at the same time.

There are powerful static analysers available such as Blast(24), available. However, for our need to ensure the complete mediation property, we don't need a framework which deals with complex types and data structures. Torvalds, creator and maintainer-in-chief of Linux has also developed a semantic analyzer for C called Sparse(27).

The mediation property which we have described in the previous section despite introducing the above mentioned constraints provides us with precise modelling.

V. Static analysis on paths

The goal of static analysis is to verify that information flow goes through a LSM hook or that it is impossible. In order to do so, we need to find the information flows for any CFG representing a system call.

A. Verifying complete mediation

We consider the set *Paths* of all paths in a CFG.We introduce two particular subsets in this:(1) the set Paths_{flows} of paths starting at the initial node of the CFG and ending at one node generating an information flow; and (2) the set Paths_{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 $\text{Paths}_{flows} \subseteq \text{Paths}_{LSM}$. However, as we will see, this is not the case. Pf = Paths_{flows} Paths_{LSM}. The sets involved in the analysis are shown diagrammatically in Fig. 5. represents the set of paths that may be problematic.

As explained earlier, some paths in Paths_{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 $P_f \subseteq I$ where $I \subseteq$ Paths is the set of impossible paths in the CFG.

The property which we are looking to verify is the complete mediation property. Explaining the entire proof is out of scope of this paper. Hence, we state the entire property here.

1) Property 1: (Complete Mediation)

Complete mediation holds iff: $P_f \subseteq I$, 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.

B. Soundness of proofs

The soundness of our analysis is stated as follows.

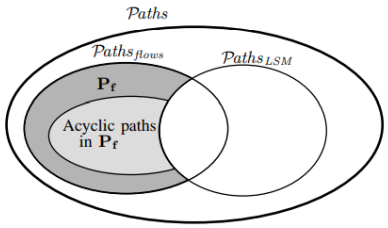

Fig. 5. Sets involved in analysis

1) Theorem 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 $k_1 \rightarrow *p\ k_2$, we have $\theta_1 \vDash k_1 \Longrightarrow \theta_2 \vDash k_2$

In simple words, according to the soundness property, executable paths are defined as the set of all paths p such that there exist two concrete configurations θ_1 and θ_2 such that $\theta_1 \rightarrow_p \theta_2$. A transition may not exist from on configuration to another if, along the path p, there is an edge bearing a condition which is not verified by the memory state. e. 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 $k_1 \rightarrow_p * k_2$, then $\not\vDash k_2$. 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.

2) Theorem 2 : (Executable vs Impossible paths)

The sets of executable paths and impossible paths are disjoint.

C. Implementation and results

The static analysis is run during the compilation itself. To this effect, we use the plug-ins developed by Georget et al. We also followed the same methodology as proposed by Georget et al. in the previous kernel version 4.3. We performed the experimentation for kernel version 4.20 which is the last stable release as of now.

We use a plugin developed by Geroget () 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 branching. 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. We make use of the tools available on the kayrebt website (31).

In order to do so, we first look at the system calls difference between kernel version 4.3 and 4.20. In total we identify 460 system calls in the stable release 4.20. But from the observation, we can also notice that not all system calls are capable of generating flows. Understanding the difference between the system calls between different kernel versions, itself is a separate study. The previous study was done by Hauser (32) for the kernel version 4.7 for his doctoral thesis. New system calls are added and used as per the applications requirements.

Prior to running analysis,we first need to place a special annotations on tplaces 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 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 [5], 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.

In order to understand and visualize the information flows, we use the tools available on http://kayrebt.gforge.inria.fr/. We categorize the flows into two types discrete flow and continuous flow as shown in the Table 2, 3 and 4. Table 2 shows a summary of the discrete flows for the different types of system calls. We have similar flows recorded for the write, send, receive system calls. Table 3 shows the system calls which were updates in the kernel version 4.20. However, Georget did a similar analysis and found the missing LSM hooks for 29 system calls which were updated from kernel version 3.2 to 4.3. Table 4 lists the system calls from version 4.3 to 4.20 whose information flows which are discrete do not trigger all LSM hooks. Table 5 shows system calls added from version 4.3 to 4.20 which do not trigger all LSM hooks when called. These system calls have continuous flows. This means that the flow is from both sides, the file to the memory and vice versa. However, we notice that the LSM hook is triggered only once during the exchange.

TABLE II
READ SYSTEM CALLS

System calls	Discrete flow
read	File → memory of the calling process
readv	File → memory of the calling process
preadv	File → memory of the calling process
pread64	File → memory of the calling process

TABLE III
UPDATED SYSTEM CALLS IN VERSION 4.20

System calls	Discrete flow
vmsplice.	Memory of the calling process → tube tube → memory of the calling process
process_vm_readv	Memory of another process → memory of the calling process
process_vm_writev	Memory of another process → memory of the calling process

VI. CAMFLOW

We have given a little introduction to Camflow in the previous sections. The Camflow[6] builds upon and learns from previous OS provenance capture mechanisms, namely PASS, Hi-Fi, and LPM. The provenance data is captured through Linux Security Module hooks and NetFilter hooks. The provenance data is transferred to the user space through relayfs, where it can be stored or analysed. Applications can enrich system level provenance with application-specific details through a pseudo-file interface. The provenance capture can be tailored to suit the needs of the application. This is done through pseudofiles and restricted to the owners of the capability CAP_AUDIT_CONTROL. Camflow provides a library that, through an API, abstracts interactions with the pseudo-files and relayfs. This is shown in Fig. 3.

CamFlow records how information is exchanged within a system through system calls. Some calls represent an exchange of information at a point in time e.g., read, write; others may create shared state, e.g., mmap. The former can easily be expressed within the PROV-DM model, the latter is more complex to model. Fig 7 gives an example of whole-system provenance capture example. We can see that Fig. 7 build on the previous work which is illustrated in Fig 4. Fig 4. displays the working of a LSM hook for an 'open' system call. Fig. 7 displays the open system call, however this time, it captures the information and then tailors the information as explained above.

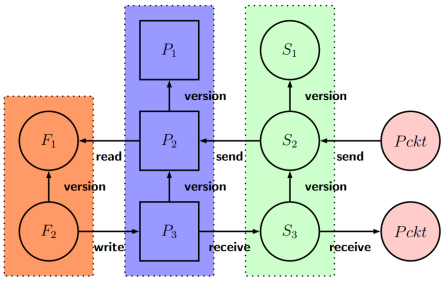

Fig. 6. An example of a provenance graph. CamFlow-Provenance partial graph example. A process P reads information from a file F, sends the information over a socket S, and updates F based on information received through S

TABLE IV
UPDATED SYSTEM CALLS IN VERSION 4.20 WHICH DON'T TRIGGER LSM HOOKS BUT HAVE DISCRETE FLOWS.

System calls	Discrete flow
migrate_pages	Memory of another process → memory of the calling process
move_pages	Memory of another process → memory of the calling process

TABLE V
UPDATED SYSTEM CALLS IN VERSION 4.20 WHICH DON'T TRIGGER LSM HOOKS BUT HAVE DISCRETE FLOWS.

System calls	continuous flow
mmap_pgoff.	Regular file or device ↔ memory of the current process regular file or device → memory of the current process

A. Provenance Data Model

As explained in the previous sections according to W3C standards the provenance is defined as a directed acyclic graph(DAG). Fig 6 represents an example of a provenance graph. A process P reads information from a file F, sends the information over a socket S, and updates F based on information received through S.

Provenance graphs are acyclic. A central concept of a CamFlow graph is the state-version of kernel objects, which guarantees that the graph remains acyclic. A kernel object (e.g., an inode, a process etc.) is represented as a succession of nodes, which represent an object changing state. We conservatively assume that any incoming information flow generates a new object state. Nodes associated with the same kernel object share the same object id, machine id, and boot id (network packets follow different rules described in 4.2). Other node attributes may change, depending on the incoming information (e.g., setattr might modify an inodes mode). Fig. 6 illustrates

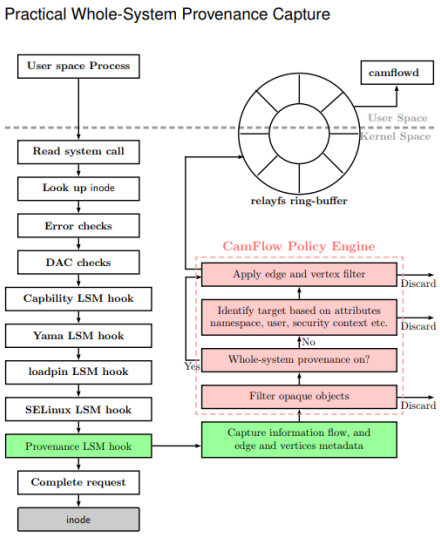

Fig. 7. Executing an open system call. In green is the capture mechanism. The pink is the provenance tailoring mechanism.

CamFlow versioning.

ig. 3 illustrates CamFlow versioning. In Fig. 6, we group nodes belonging to the same entity or activity (F, P and S), based on shared attributes between those nodes (i.e., boot_id, machine_id, and node_id). In the cloud context, those groups can be further nested into larger groups representing individual machines or particular clusters of machines. For example, when trying to understand interactions between Docker containers we can create groups based on namespaces (i.e., UTS, IPC, mount, PID and network namespaces), control groups, and/or security contexts. The combination of these process properties differentiate processes belonging to different Docker containers. Provenance applications determine their own meaningful groupings.

B. Capture mechanism

Provenance observed at the system layer may not be sufficient for all use cases. It may be necessary to disclose to the provenance infrastructure the inner workings of a process. PASS was the first system to introduce the possibility for an application to disclose provenance to complement provenance observed at the OS level, and we implement a similar mechanism. However, this process is complex and generally requires engineering effort. To accommodate legacy applications we provide an additional mechanism to annotate the system provenance graph with log information (as illustrated in Fig. 2). Past research has demonstrated that internal provenance of, for example, web-servers can be derived from their logs. Therefore, we can understand an applications inner workings without complex modifications.

C. Provenance graphs show and tell

The original goal is to eventually make sure that any actions made by a process and that have influence on the system are represented in the graph. The reason we want this to be the case is so that we can detect violations. Based on the previous analysis we have shown that upon adding two new hooks, we get the assurance that all information flows pass through LSM hooks. According to the example in Fig 7, we observe that CamFlow is stacked on all the different modules, which are the Capability LSM hook, Yana LSM hook etc. Every implementation captures different flows and we have explained this in the previous sections. The provenance LSM hook is the lowest in the stack after all the policy stacks. Based on the architecture proposed by Camflow, if the information flows pass through the LSM hooks, the data is indeed getting captured. If all the data is being collected, and no selective capture mechanisms as explained before are being employed we can make get the surety that it shows up on the graph. This is the first part of this part.

1) Property 2: (Whole system provenance)

Iff the Property 1 holds, which means that all information flows pass through the LSM hooks, then CamFlow captures all the data.

2) Theorem 3: Transitivity

If an implementation Γ_1 adapted for LSM captures information γ and another implementation Γ_2 is a superset of Γ_1 then Γ_2 captures the information γ.

In other words this theorem says that if in SELinux all information flows pass through LSM hooks, which holds if the Property 1 holds. Then, since CamFlow is a direct stackable module on top of all existing modules, it will capture all the flows which SELinux will. The next Theorem 4 is a followup which is comes after the Theorem 3.

3) Theorem 4: Module over multiple implementations

If a set of implementations Γ_1 Γ_2 Γ_3 ... Γ_n are adapted for LSM captures information γ_1, γ_2 .. γ_n and another implementation Γ_F is a superset of Γ_1, Γ_2 Γ_3 ... Γ_n then Γ_F captures the information γ_1, γ_2 .. γ_n.

This theoretically shows that if LSM interface makes sure all flows pass through LSM hooks, then CamFlow indeed does capture all the data. The second part to this formalism would be the property 3.

4) Property 3: Provenance Graphs

According to this property if Property 2 holds, and CamFlow captures all the data successfully, then the graph reflects the relations and the properties.

We can use theorems similar to Theorem 3 and 4 to verify the property 3.

VII. Limitations

The limitations of our approaches are that we ignore the side channel attacks. We do not consider timing attacks. We restrict ourselves to explicit information flows such as information storing or interprocess 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.

VIII. Related Work

There are many types of tools dedicated to the analysis of the kernel. The most basic method in order to find errors faster, is to look for reasons in the code known to be symptomatic of certain classes of problems in the kernel.

A. Information flow control systems

Previous work on information flow control enforcement at the OS level, such as HiStar [41], Flume [42], and Weir [43], uses labels to define security and integrity contexts that constrain information flows between kernel objects. Labels map to kernel objects, and a process requires decentralised management capabilities to modify its labels. Point-topoint access control decisions are made to evaluate the validity of an information flow. Through transitivity, it is possible to express constraints on a workflow (e.g., collected user information can only be shared with third parties as an aggregate). SELinux [44] provides a similar information flow control mechanism but without decentralised management. A typical way of representing and thinking about information flow in a system is through a directed graph. However, current object labelling abstractions do not take advantage of this representation, and it is difficult to reason about when defining policies. CamQuery differs from these systems in that it allows the implementation of such mechanisms directly on the graph abstraction.

IX. Conclusion

Hence, finally we run the analysis which was introduced by Georget et al[3] for the version 4.3 and we find three system calls which call LSM hooks but not all paths are mediated. These are listed in table 4 and 5.

We further provide a formalism to show that if all the flows pass through LSM hooks,then the flows gets captured by CamFlow. If they get captured by CamFlow, and assuming that no data is lost during the construction of the graphs, all the anomalies show up in the graphs.

X. Future work

The next step would be doing a similar analysis for the kernel version 5.x. Automating the technique to utilize the tools for the newer kernel version should be done. Georget et al.[3] provided a set of tools which when used, help in visualizing information flows, performing static analysis and finally getting to the results if any LSM hooks are missing. If we can automate this process for the future kernel versions, it will be a great contribution.

Another contribution can be analysing the different IDS utilizing LSMs in different forms such as Blare, Weir, Laminar and understanding how and why they differ if they are utilizing the same underlying mechanisms. Is it just about the implementation or is there more to offer?

If indeed every action we perform shows up in a provenance graph, we can use that information to recreate information. I do mention this in the paper, however, proceeding in that direction sounds interesting.

Another challenge would be to consider covert channels during the analysis.

I think health care could benefit a lot from such approaches. If we can have such concept in inside the human body, which creates a provenance of everything the body consumed. This is probably fiction for now, but maybe someday.

References

[1] Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, Andy Hopper, an "A Primer on Provenance," acmqueue, 2014.

[2] Daniel Crawl and Ilkay Altintas , A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows.

[3] Laurent Georget, Mathieu Jaume, Guillaume Piolle, "Verifying the reliability of operating system-level information flow control systems in linux," Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering, FormaliSE 17, pages 1016, Piscataway, NJ, USA, 2017. IEEE Press.

[4] GERWIN KLEIN, “Operating system verificationAn overview,”Sadhan a Vol. 34, Part 1, February 2009, pp. 2769. Printed in India

[5] Jonathan Pincus and Brandon Baker , “Mitigations for Low-Level Coding Vulnerabilities: Incomparability and Limitations ,” 2004.

[6] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, Margo Seltzer, “Runtime Analysis of Whole-System Provenance ,”16 pages, 12 figures, 25th ACM Conference on Computer and Communications Security 2018.

[7] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, Jean Bacon, Practical Whole-System Provenance Capture , SoCC ’17 Proceedings of the 2017 Symposium on Cloud Computing.

[8] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo Seltzer, Harvard University
, “FRAPpuccino: Fault-detection through Runtime Analysis of Provenance ,” HotCloud’17.

[9] PASQUIER, T. F.-M., SINGH, J., BACON, J., AND EYERS, D. , “Information flow audit for paas clouds. Cloud Engineering (IC2E), 2016 IEEE International Conference on (2016), IEEE, pp. 4251.

[10] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. , “ Provenance-aware storage systems. ,” In USENIX Annual Technical Conference, General Track (2006), pp. 4356..

[11] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K , “Hi-fi: collecting high-fidelity whole-system provenance ,” In Proceedings of the 28th Annual Computer Security Applications Conference (2012), ACM, pp. 259268.

[12] Adam Bates, Dave (Jing) Tian, and Kevin R.B. Butler , “Trustworthy Whole-System Provenance for the Linux Kernel. 24th USENIX Security Symposium, 2015.

[13] PASQUIER, T. , “Camflow information flow patch. In https://github. com/CamFlow/information- flow- patch. .

[14] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman. , “Linux Security Modules: General Security Support for the Linux Kernel. Proceeding Proceedings of the 11th USENIX Security Symposium Pages 17-31
August 05 - 09, 2002 .

[15] PASQUIER, T. , “CamFlow development. In https://github.com/CamFlow/camflow-dev.

[16] INRIA , “The Kayrebt Toolset In http://kayrebt.gforge.inria.fr/

17 Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In International Conference on Database Theory. Springer, 316330.

18 Allison Woodruff and Michael Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In International Conference on Data Engineering. IEEE, 91102.

19 Khalid Belhajjame, Reza BFar, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, Luc Moreau, and Paolo et al. Missier. 2013. Prov-DM: The PROV Data Model. Technical Report. World Wide Web Consortium (W3C). https://www.w3.org/TR/prov-dm/

20 Rastislav Bodk, Rajiv Gupta, and Mary Lou Soa. 1997. Refining Data Flow Information Using Infeasible Paths. SIGSOFT Software Engineering Notes 22, 6 (Nov. 1997).

21 Richard Matthew Stallman and the GCC developer community. 2013. Using the GNU Compiler Collection (GCC). Technical Report. https://gcc.gnu.org/ onlinedocs/gcc-4.8.4/gcc/

22 author=”Xiao, Jidong and Huang, Hai and Wang, Haining”, editor=”Thuraisingham, Bhavani and Wang, XiaoFeng and Yegneswaran, Vinod”, title=”Kernel Data Attack Is a Realistic Security Threat”, booktitle=”Security and Privacy in Communication Networks”, year=”2015”, publisher=”Springer International Publishing”/

23 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/

24 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.

25 WireX Communications. Linux Security Module. http:// lsm.immunix.org/, April 2001.

26 Stephen Smalley, Timothy Fraser, and Chris Vance. Linux Security Modules: General Security Hooks for Linux. http: //lsm.immunix.org/, September 2001.

27 Neil Brown. Sparse : a look under the hood. Linux Weekly News. 8 juin 2016. url : https://lwn.net/Articles/689907/ (visit le 01/02/2017) (cf. p. 29).

28 Yoann Padioleau, Julia L. Lawall, Ren Rydhof Hansen et Gilles Muller. Documenting and Automating Collateral Evolutions in Linux Device Dri- vers . In : European Conference on Computer Systems (EuroSys 2008). Glas- gow, Scotland : ACM, avr. 2008, p. 247260 (cf. p. 29).

29 Junfeng Yang, Ted Kremenek, Yichen Xie et Dawson Engler. MECA : an extensible, expressive system and language for statically checking security properties . In : ACM conference on Computer and Communications Security (CCS 2003). Washington D.C., USA : ACM, oct. 2003, p. 321334 (cf. p. 29).

30Jeffrey Scott Foster. Type qualifiers : lightweight specifications to improve software quality . Thse de doct. Berkeley, CA, USA : University of California at Berkeley, 2002 (cf. p. 29).

31 http://kayrebt.gforge.inria.fr/

32 Christophe Hauser. Dtection dintrusion dans les systmes distribus par propagation de teinte au niveau noyau . Doctoral thesis. Rennes, France : University of Rennes 1, juin 2013

33] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-Aware Storage Systems. In Proceedings of the 2006 USENIX Annual Technical Conference, 2006.

34 J. Seibert, G. Baah, J. Diewald, and R. Cunningham. Using Provenance To Expedite MAC Policies (UPTEMPO) (Previously Known as IPDAM). Technical Report USTC-PM-015, MIT Lincoln Laboratory, October 2014.

35 A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. In NDSS Workshop on Security of Emerging Network Technologies, SENT, Feb. 2014.

36 A. Gehani, B. Baig, S. Mahmood, D. Tariq, and F. Zaffar. Finegrained Tracking of Grid Infections. In Proceedings of the 11th IEEE/ACM International Conference on Grid Computing, GRID10, Oct 2010.

37 D. Tariq, B. Baig, A. Gehani, S. Mahmood, R. Tahir, A. Aqil, and F. Zaffar. Identifying the Provenance of Correlated Anomalies. In Proceedings of the 2011 ACM Symposium on Applied Computing, SAC 11, Mar. 2011.

38 D. Nguyen, J. Park, and R. Sandhu. Dependency Path Patterns As the Foundation of Access Control in Provenance-aware Systems. In Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance, TaPP12, pages 44, Berkeley, CA, USA, 2012. USENIX Association.

39] Q. Ni, S. Xu, E. Bertino, R. Sandhu, and W. Han. An Access Control Language for a General Provenance Model. In Secure Data Management, Aug. 2009

40 J. Park, D. Nguyen, and R. Sandhu. A Provenance-Based Access Control Model. In Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST), pages 137144, 2012.

41. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazires. 2006. Making information flow explicit in HiStar. In Symposium on Operating Systems Design and Implementation (OSDI06). USENIX Association, 263278.

42 Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In ACM SIGOPS Operating Systems Review, Vol. 41. ACM, 321334

43 Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC enforcement on Android. In USENIX Security Symposium. 11191136

44 Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. NAI Labs Report 1, 43 (2001), 139.