Proper information flow monitoring for systems Linux operating

Laurent Georget

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale 601 « Mathématiques et Sciences et Technologies de l'Information et de la Communication (MATHSTIC) »

présentée par

Laurent GEORGET

préparée à l'unité de recherche IRISA (UMR 6074) au sein de l'équipe-projet commune CIDRE — Inria / CNRS / Université de Rennes 1 / CentraleSupélec

---

Suivi de flux d'information correct pour les systèmes d'exploitation Linux

Thèse soutenue à Rennes le 28 septembre 2017

devant le jury composé de :

Jean GOUBAULT-LARRECQ
Professeur des universités, ENS Paris-Saclay / rapporteur

Marie-Laure POTET
Professeure des universités, Ensimag – Grenoble INP / rapporteur

Erven ROHOU
Directeur de recherche, Inria / examinateur

Sarah ZENNOU
Ingénieure de recherche, Airbus / examinatrice

Mathieu JAUME
Maître de conférences, UPMC / directeur de thèse

Valérie VIET TRIEM TONG
Professeure associée, CentraleSupélec / directrice de thèse

Thanks

Many individuals and institutions contributed to the completion of three years of scientific production? That materialized by this thesis and publications page 181 ; This page is dedicated to them.

First of all, I wish to thank the members of my jury who agreed to evaluate this work: Mr Erven Rohou, who chaired the jury, which had previously chaired the jury of my defense midterm; Mr Jean-Goubault Larrecq, which reported this thesis, whose many questions, forcing me to clear my mind, helped me to better present this work; Marie-Laure Potet also rapporteur of this thesis; Sarah Zennou.

I thank my directors and thesis supervisors, who all contributed both scienti? Cally and humanly this doctorate. Mathieu in the role of semantician, contributed enormously to the formalization and evidence of contributions of this thesis. I also thank him for working sessions at Jussieu and bo buns next to the arenas of Lutèce, and its inordinate enthusiasm for style? Arrows LaTeX. Valerie who directed my thesis locally provided in this thesis the overview and history of the project and its Blare model. I want to thank her for the time she was able to devote myself to the good times and the freedom of action that my independent nature, or wild, requires. Our culinary experiments and para-culinary provided the relaxation times required that this work expires. Frédéric framed this thesis providing expertise on the Linux kernel and suggesting what make intelligible the formalization proposed by Mathieu and me. I thank him for having always been interested and enthusiastic about respect ideas and implementation techniques that I have proposed. I will keep the memory of moments to fix bugs or administer capricious servers. In? N, Guillaume provided each critical step of this thesis advised his attention his corrections and comments on my writing. I owe him a lot regarding the quality of my publications. I also thank him for his clarifications and his great historical culture and the law which my coffee breaks bene? Ated.

Thank the dark heart of my family and friends who have given me their help and supported me during this thesis. Benoit and Camille, in particular, have hosted me in Paris whenever I've needed. My office colleagues, and those who have gone since I began this thesis, were the companions of countless discussions desultory, parts billiards and pints. I thank my parents who always pushed me to go further and have shown me unwavering support in everything I undertook, in terms of education, my professional or personal life. In? No, thank Dolly, whose presence at my side is precisely what he always took me. Besides my wife, this is the intellect with which I can share most directly.

Contents

Summary (in French) & Abstract (in English)

summary

We seek to improve the state of the art flow control implementations of information in Linux systems. The control? Ow of information is to monitor how information is disseminated in the system once out of its original container, in contrast to access control that can help to enforce rules on the how the containers are scanned.

The first contribution of this thesis is a plugin for the GCC compiler to extract the graph? Ow control of core functions. These graphs are used to support the static analysis as visualization.

Then the question arose whether the framework of the Linux Security Modules which is used to implement the monitors? Ow of information is implemented so as to enable the capture of all greeting products by system calls. We have created and implemented a static analysis to address this problem. This static analysis, the correction of which has been proven with Rooster, has allowed us to extend the LSM framework to capture all the stream?.

In? No, we found that monitors vows today were not resistant to the conditions of competition between the vows and could not treat some open channels such as projections? Files in memory and shared memory segments between processes. We implemented Rfblare, a new tracking algorithm of greeting, conclusively proven with Rooster.

Abstract

We look forward to Improving the implementations of information? Ow Control Mechanisms in Linux Operating Systems. Information FlowControl AIMS at monitoring how information disseminates in a system once it is out of icts original container, access control Unlike Merely qui can apply one rule how the containers are accessed.

The? Rst contribution of this thesis is a plugin for the GCC to compile reliable extract the control? Ow graphs of the Linux kernel functions. These graphs can be used as a basis for static analyzes and visualizations.

SECONDLY, we Studied the Linux Security Modules framework qui is used to Implement information trackers ow to answer an open problem: the Framework is Implemented in Such a Way That All ows generated by system calls can be captured? We-have created and Implemented static analysis to address this problem and Proved

Correction icts with the Coq proof assistant system. This analysis allowed us HAS to Improve the LSM framework in order to capture all? Ows.

Finally, we-have Noted That information current? Ow trackers are vulnerable to race condition entre? Ows and are Unable to Cover Some overt channels of information Such as? The mapping to memory and shared memory segments entre processes. We-have Implemented Rfblare, a new algorithm of? Ow tracking. The adjustment of this algorithm has-been Proved with Rooster.

A process can not be Understood by stopping it.
Understanding must move with the? Ow of the process,
must join it and? Ow with it.

Frank Herbert, *Dune* 1965.

Chapter 1

Introduction

Many theses on IT security begins with peeling Special Rapporteur how, now that we live in the information society, we become increasingly unable to do without computer skills and how to become associated threats increasingly sophisticated, which calls for appropriate countermeasures. In 2017, these introductions are really outdated and n'e? Raient anyone. However, not having a lot more imagination than another, I just do exactly the same type of grip, albeit in a roundabout way.

The work we have undertaken in this thesis therefore focus on information security, and more specifically on the context, that of the Linux operating systems using a core and two security properties to ensure throughout the life of the system.

con? dentiality Only authorized users should have access to information deemed sensitive (that classi? cation emanating from the Act, a supercilious system administrator or other users concerned with privacy). If a user is able to read the content of a? Private shit another user, for example with the complicity of third parties, if a hacker manages to ex? Lter last? Lm still in post-production of a studio or if a doctor sends via their personal mail folders of patients, there is a breach, respectively, of the privacy of the user, the trade secret of the studio and patient confidentiality.

integrity The information stored in the system should not undergo change? - cations making them lose their value to the system or users who are its owners. The accidental crash of a PhD thesis in progress, the introduction of transactions dating back to 2016 in the account book from 2017 or the unauthorized creation of a user account in a service by a hacker are three examples of corruption intolerable, respectively touching the folder in which was contained the manuscript, account book and user database.

To meet the needs of con? Dentiality and integrity, system administrators de? Ne of *security policies* assigning each user *permis- sions* and each object of the system may contain an information *security level.* These policies express security needs but know? Not feel in the

meet without the mechanisms for their implementation e? ective. This thesis focuses on one of these mechanisms: monitoring flow of information?.

The most common way to meet the security objectives is access control. For this, is associated with each level of security permissions that are required to read or modi? Cation of objects assigned to this level. In this way, we ensure that only properly authorized users to read or alter information when stored in a? Le, or other object that can store information, marked the appropriate level of security. However, the fragility of the approach lies in this: once the information out of their original container, the policy can no longer protect them. To fully protect the information, the policy must be transitive: if Alice has access to a file and not Bob, Alice must not have means to communicate with Bob, otherwise it could pass him, even accidentally, the data? le. That would violate the property of con? Dential system.

The control? Ow of information responds to this problem. By keeping 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, not limited as communications in the case of access control. In the above example, Alice has the right to contact Bob, until it takes knowledge of the content of the? Le, after which it must stop contacting Bob? No not to fuiter information secret.

Naturally, of greeting information control mechanisms are much more complicated to implement and there is not, at present, de solution? Nal to this problem. As part of this thesis, we focused on how to implement the tracking? Ow of information on Linux. ? Monitoring flow of information is a crucial step of the control flow information: it keeps the history stream in the system?. The monitoring tools? Ow of information that we studied are all implemented in the Linux kernel using the *framework*

Linux Security Modules (LSM)*. However, this framework was originally designed for the implementation of access control and not of vows, and the question arises whether it is appropriate for the task. ? In addition, even if a controller ow of information could intercept all the individual flow in the system, it still has to deal with an additional problem: the conditions of competition between ux may mask some of them to? eyes ux? monitor. We propose in this thesis means to check first with formal methods a necessary condition on monitoring ows: to control all individual flow in the system??? and a sufficient condition: given a certain list of ux followed individual, able to observe any composition of these, even under conditions of competition?.

The organization of this thesis is as follows. The chapter 2 is a state of the art control? ow of information in the operating systems and the means to implement it and the means to analyze and veri? er properties on the Linux kernel and especially the *framework* of the *Linux Security Modules*. The chapter 3

details the concept of "object of the system may contain information" we have deliberately left vague in this introduction. We present what the di? Erent Linux kernel information containers and how abstractions such as? Les, the *sockets* networks, shared memory or processes are implemented by internal data structures

*. Terms followed by an asterisk on their first occurrence are de? Ned in the glossary page 189 .

core. In chapter 4 We present our first contribution is not directly linked to the monitoring? Ow of information but the Linux kernel analysis. In e? And we attacked in this thesis of dice? S scienti? C, using formal methods and theoretical tools, and we have also faced software engineering technical problems. The Linux kernel code base is particularly large, complex and rapidly changing. Our first task was therefore to understand its organization to take control of its code and propose models to describe, analyze and visualize. The chapter 4 consists of the description of Kayrebt project, a suite of tools to extract the kernel of the operating system call graph and? ow control and a visualization GUI. In chapter 5 We present our second contribution, dealing with the necessary condition expressed above. We list the Linux kernel system calls and we offer a static analysis for veri? Er the framework LSM enables both a monitor? ow of information implemented using it to observe the flux caused by these system calls. In? N, Chapter 6 We present our third contribution, a response to the sufficient condition expressed above as a new tracking algorithm ow accomplishing two goals:? Withstand the conditions of competition and follow the stream due to projections? files and shared memories, information containers that are, at present, no suitably treated monitor greeting information implemented for Linux. We show first vulnerability monitors greeting implemented for Linux exposing several attacks exploiting the same flaw against all these monitors, yet independently developed. The correctness of our new algorithm is demonstrated with the proof assistant Coq. We show the practicality of our approach by implementing our new algorithm in the form of *Rfblare,* a new module LSM . The attacks that we have developed for e? Ectuer of greeting information escaping the other monitors are ineffective against *Rfblare*.

Chapter 2

State of the art

By studying the literature on the control? Ow of information in UNIX, and Linux in particular, we came across implementations spanning the entire period of the last fifty years. The principles of control? Ow of information are e? And very studied, but the details of its implementation are still an active research subject. In the particular context of the Linux kernel, we then looked at ways and risks of the interposition in the system calls to ensure security properties, and in our case, take vows of information. In? N, as we looked at ways to build formally correct implementations of monitors greeting for Linux

2.1 Access control and control? Ow of informa- tion

The control? Ow of information is a security mechanism consisting in over-watch how information spreads in a system, what are the vows of information that take place, a? N to have a accurate view of information that each container contains. This information is used to classify each container at a certain level of security and implement security policies to control access to information system users based on their accredi- tation. Depending on the context, the concepts of *system,* of *Information container* and of *information ows* may vary widely.

The control? Ow of information in the operating system was born from a need to formalize the systems security objectives, and propose policies to achieve these objectives. Mandatory access control has been an important research topic from? N 1960 when the *Department of Defense (DoD)*

unien-states began to look for ways to apply the same classi? cation and guarantees con? dentiality in computer systems in the rest of its archives and communications.

These needs have inspired *System Development Corporation* who developed ADEPT-50, a system having a new security model known subsequently as *High Water Mark,* and exposed in an article by Weissman [102]. This system considers four types of objects: users, files, devices and sessions?. Each

object is associated with a *pro? s security* compound of *authority,* a certain level of sensitivity caught in a scale of values, a set of *categories* which are a non-hierarchical partition di? erent information topics contained in the system and a *concession* which is the set of privileged users that can access the object. The concession is de ned as?:

- when it comes to a user, that user himself by convention;
- when it comes to an end, all users can connect to it;
- in the case of a session, the user having initiated;
- and? No, when it comes to a? le, all users allowed to consult as the requested access.

ADEPT-50 considers two types of access, read-only and read-write, and ow control mechanism also includes notable feature, the locking mechanism: several readers can access the same file simultaneously, but? writers get exclusive access.

The deWeissman work thus provides a de? Nition of the concepts of formal *classi - cation* and of *permission* liberally used by the DoD in its standards and procedures: *classi? cation* an object is the pro? s minimum security necessary to access the object and *permission* is the pro? s maximum security that can endorse a user. These de? Nition were included in all security models inspired by the DoD . The interesting element in this model is the distinction made between the user and session. Every user has a pro? S security? Xed, depending on its functions. However, when connecting to the system, the session is not open with maximum duties, but rather with the smaller pro? S security satisfying the terminal to which it is connected (all terminals do not allow the access the same information system and launch the same commands). Thereafter, every action done in the session, if the pro? S safety it is below the required permissions, it is increased to the minimum level to satisfy access within the limit level the user. When? Le is created or modi? Ed, the pro? S does not become the user but of the session. This keeps the information in a more "public" status and can not unduly restrict access to information. It is this mechanism that does the name *High Water Mark* that evokes the maximum height reached by a river during a flood. The brand change things? Ed is the highest mark of previously viewed items during the session. Thus, if a general connects to the canteen terminal to order a sandwich, his order will be met by the canteen service. But if he does it from the terminal in his office on which he consulted a report on the Soviet nuclear arsenal (or North Korea, the twenty e century), only his loyal aide will know? ciently empowered to bring him his meal tray.

About the same time, Bell and LaPadula proposed a multi-level access control model [52]. In this model, the process (or *topics)* just like *objects* system (mainly? le but the model is voluntarily vague on this point) have a level of sensitivity. In the case of process, it is the accreditation of the owner. The model describes two types of actions of subjects on objects: reading and writing. A process can read an object if its accreditation is greater than or equal to the sensitivity of the subject; conversely it can not write to it if its accreditation or less. In applying this policy, it is clear that no low-accreditation process can not access a

Information from an object of high sensitivity, even working with a "traitor" high level. The notion of control? Ow of information is present. Unlike policy *High Water Mark* the level of a subject or object can not change: An action is either allowed or prohibited but it does not modify the level of any subject or object?.

The seminal work of Denning [20] Were the first to introduce explicitly the concept of greeting information and formalize a model control policies of these vows. The first contribution of Denning is a relationship *can? ow* between safety classes. Safety Classes are abstract objects partition a system. In the case of Bell-LaPadula model, it would be such sensitivities objects. The relationship *can? ow* describes what the? ow of information allowed in the system. Two classes are related if allowed to copy information from one object of the first class in a subject of the second. This relationship is actually a preorder:

- It is re? Ective, so it is always allowed a container *preserve* the information it already has.
- It is transitive, because otherwise it would be possible to make greeting illegal composing greeting legal.

The original pattern further added the anti-symmetry, so transforming the preorder partial order, arguing that if two classes of information may freely exchange information while one of the two classes is redundant. All these conditions - added to the reasonable assumption that in a system there is a finite number of security classes, for administrative convenience - make it possible to describe the policy as lattice •. In formal cemodèle, it becomes possible to express many pre-existing policies in a common formalism and compare. The argument of the article is based on the assumption that the mechanisms meant to enforce security policies can consider only the class of objects concerned when accessed, whether the authorize or prohibit. In particular, it is not considered that ux? Past the system are taken into account, except by changing disappointed nitely safety class objects to restrict vows possible. It is this element that is challenged by the tracking mechanisms? Ow of information, whose intuition reflected in ADEPT-50 [102].

The work of Jaume and his staff, in particular, address the di? Erence between the access control and control? Ow of information [44 - 46]. In theory it develops, Jaume present access control as a property *states* a system while controlling greeting covers *executions* in the system. More specifically, the access control promises that it is not possible to be in a state where information from a source of some safety class *AT* is in an information container of a class *B* while no subject has permission to read the contents of a source *AT* modi has permission? er a destination *B* and conversely, any subject having permission to modify? er a container *B* has the right to read from a source *AT*. In the case of the control flow of information, it is guaranteed that nothing that matters can not be can cause greeting a class *AT* to a class *B* if this greeting is against the policy. As seen, the motivations are similar but the implementation is di? Erent. access control policies built on the model of Denning automatically guarantee the control? ow of information [20]. However, it is possible to

ensure control of vows other than the access control, and so more strictly? do.

In e? And we can consider for example the following access rights matrix, extracted from "Flow of interpretation based access control" [46].

	o_1	o_2	o_3	o_4
Alice	read, write		read	
Bob	read	read, write		
Charlie		read, write		write

Suppose that is associated with each container o_i in this example the safety class C_{oi} and Alice, Bob and Charlie (or better said, the process launched by those users) classes respectively *A, B* and *C.* One can derive the table indicating the relation permits access.

$C_{o1} \rightarrow AT$ $AT \rightarrow C_{o1}$ $C_{o3} \rightarrow AT$

$C_{o1} \rightarrow B$ $C_{o2} \rightarrow B$ $B \rightarrow C_{o2}$

$C_{o2} \rightarrow C$ $C \rightarrow C_{o2}$ $C \rightarrow C_{o4}$

$AT \rightarrow AT$ $B \rightarrow B$ $C \rightarrow C$

$C_{o1} \rightarrow C_{o1}$ $C_{o2} \rightarrow C_{o2}$

$C_{o3} \rightarrow C_{o3}$ $C_{o4} \rightarrow C_{o4}$

This relationship is naturally closed by re? Exivité because it is impossible to ensure that a container can not contain its own content. There is, however, that this relationship is not transitive. There is no such $AT \rightarrow C_{o2}$ while Alice is able to write to the? le o_1 Bob can then read to copy the contents in o_2. In the model of Denning, this policy is incoherent because $<\{A, B, C, C_{o1}\ C_{o2}\ C_{o3}\ C_{o4}\} \rightarrow>$ not a partially ordered set. Now consider the following two executions that di? Erent than the order of executed instructions.

1. *Alice* reads from o_3	1. *Bob* reads from o_1
2. *Alice* writes to o_1	2. *Alice* reads from o_3
3. *Bob* reads from o_1	3. *Alice* writes to o_1

In both cases, all individual access are allowed. The first run has a greeting illegal C_{o3} towards B via AT then C_{o1} and therefore any model correct security must prevent it, but the second contains only greeting legal. Now, the second execution is permitted by any Denning type of policy because, to be transitive, it would require to have C_{o3} 6 $\rightarrow B$ although there is no? ow C_{o3} at B

in *this* execution. The mechanisms based on access control are therefore too restrictive in the sense that they prevent entire classes of executions causing no greeting illegal but impossible in transitive policies. of greeting information tracing mechanisms designed to solve this problem by distinguishing the two cases of execution above. In e? And here we see, what the problem is that the greeting o_3 towards o_1 via Alice place *before* the greeting o_1 to Bob, while in the second run, it takes place *after,* and in? uence so not what Bob reads from o_1. It is therefore known? Cient to a safety mechanism to keep for each container

historical information of greeting past having in? uenced its content in order to distinguish the executions featuring greeting illegal for those not featuring.

There are actually two types of access control, discretionary and mandatory. discretionary access control policies consist of a matrix such as that described above. Each user administers completely and in total freedom the access rights of its own? Les and other containers. The mandatory access control on the other hand limits the freedom of users and allows the system administrator - sometimes known as the o cial security in the literature - ask irrevocable access restrictions on some containers. Users can add additional restrictions but can not remove the restrictions set by the administrator. It is therefore possible for the administrator as part of the mandatory access control to implement strong policies,

The main di? Erence between mandatory access control and control of greeting is that access control is stateless. It can not guarantee the absence of greeting illegal by restricting the applicable policies. In contrast, the control? Ow of information can apply more policies while maintaining the lack of greeting illegal guarantees, the price of a state to maintain. For example, the control? Ow of information allows a user to read data from a certain level of sensitivity without allowing it to di? User in turn, while the access control would be to authorize access and di? usion or to prohibit the both. The discretionary access control, meanwhile, would be to apply non-transitive policies and without keeping history on greeting past. It only allows the application of simple policies and fails to guarantee no greeting illegal in the general case. He knew? T such an unauthorized process to corrupt or deceive an authorized process for ex? Lter data con? Dential.

2.2 Policy Templates and tracing greeting

As explained above, apply a stream of? Control policy informa- tion broader than those permitted by the access control required to maintain a history of greeting passed through the system. However, this history does not have to be a complete breakdown of all greeting individual occurring. He knew? T know, for each data container, which safety classes who participated in at least one greeting to this container. This lets you know what kind of information may fuiter during a greeting from that container. It is therefore to abstract the flow?.

Propagation color

How far from the most common to maintain that knowledge is *hue spread,* or *taint tracking* in English. This approach is to attach to each information container a *label,* a metadata indicating what security classes in? uenced the current contents of the container. This label is propagated along the data when a greeting occurs from one container to another container and the destination of the label is updated to re? Ect the fact that its content was changed? Ed by source. The term *propagation hue* has this mechanism. It is as if the containers were marked with a lick of paint and containers participating in? Ow of information is "stained". At any time of the

life of the system, the color of a container allows tasks to who touched him. This concept of marking dates back at least to the seminal work of Fenton [29]. In reality, Fenton explained the idea of the spread while dismissing it in its first model. The brand associated with the container, the registers of an abstract machine model in his case, are static. The merits of the spread were thus rediscovered later, including McIlroy and Reeds [56] As detailed in section 2.3 .

The concept of color spread is very generic and has been adapted for many uses. There may be mentioned in particular the Perl language which has a mechanism for marking data from untrusted sources (user input, basically) and prevent it from being evaluated as code, or as a query database [70 , 80].

Terminology

All authors have introduced similar terminology but nevertheless lies subtly di? Erent to refer to the concepts of control? Ow of information. In this state of the art, we do not claim uni? Er all the terminologies and notations but we still simplified? Ed the speech using the same word for the same concept as as was possible. Here's a glossary of these terms.

security Security is an abstract property of a system in which each model
combines disappointed formal definition. conventionally associated with it con sub-properties? dentiality, integrity and availability [79]. Under the control? Ow of information, only the con? Dentiality and integrity are discussed in the models.

con? dentiality The con? Dentiality is impossible for any user of a
system to access data that does not have permission to view. We consider only the active violations of the con? Dentiality, involving an attacker.

integrity Integrity, as regards the control? Ow of information, repre-
feel unable for any user of a system in? uence, alter or delete data that is not allowed to change? er. In the same way as before, we do not consider accidental damage to the infor- mation, such as changing the value of a bit due to an unfortunate cosmic radiation.

process A process is an active entity of the system, executing code. according to
cases, it may be a single? s execution (*thread*•) or a process in the UNIX guide (essentially, a group of *threads* Depending on the model, but this is a detail that will be relevant in this thesis that from chapter 3).

object An object is an abstraction of the system which may contain information and
can be the source or destination of a greeting. We also use the term information container.

label A label is a metadata attached to each object indicating what his
level designed Confidentiality or integrity, or having in any way whatsoever the information su? cient to allow the? control mechanism flow information to determine the legality of greeting information.

tag Present in certain models, identification tags? Ent each category or
original source system information. As such, they do not represent a

level of con? dentiality or security, but they make up the labels, which the models give a semantics in terms of levels of con? dentiality and integrity.

declassi? cation The *declassi? cation* is the step to reduce the level of security of an object. Increase the level of con? Dentiality of an object does not cause risk to the con? Dentiality, it only makes the least accessible object. However, the declassified? Cation, which is the inverse operation poses a risk. If the object contains very sensitive data, they may be exposed to unauthorized individuals, so this is an operation requiring privileges.

approval The *approval, endorsement* in English, is the counterpart of the declassified? - cation in terms of integrity. This is not a security risk to diminish the integrity of an object, but instead increase the integrity is a privileged operation because if the object contains corrupted data or that are not con? Dence, they may contaminate other con? dence objects. It is this operation called approval.

2.2.1 Several levels of application tracing

The control? Ow of information can be applied to many floors in an operating system or between network operating systems. In e? And, depending on what is considered to be a *information container,* and according to the greeting that is to be observed, we can distinguish several levels of granularity.

level equipment The vows are traced throughout the system with the help of equipment, by example directly into the processor. The vows are caused by machine instructions and information containers are the processor registers and memory words. For example, a statement such as ADD EAX, EBX, which means? erait "add registry EAX the register contents EBX "Produce a? Ow EBX at EAX. SEL4, View, RIFLE, or Loki apply this type of tracking very low [62 , 99 , 108 , 111].

level program The vows are drawn within a process, or a group process as the Java virtual machine. The vows are caused by instructions of the programming language and containers are the variables manipulated by the program and its input-output, such as descriptions tors? Files it handles. Some famous examples of implementations are JFlow or Jif [64 , 66], The Working Genaim and Spoto [32] And JBlare [40] For the Java language, FlowCaml [72] For ML.

level operating system The vows are drawn by the kernel in all pro- cess out of the nucleus. The vows are caused by system calls and contained neurs are abstractions offered by the operating system such as files, processes, network sockets, etc. Examples of implementation are very numerous in this category [10 , 16 , 33 , 51 , 67 , 76 , 100 , 109] And we describe in the section 2.3 .

One can of course imagine where additional levels within a network, computers, databases, network mounts, etc. constitute neurs contained and where the tracing of greeting is e? ectué by the network. Hauser example describes a mechanism for propagating shades between machines in the packets

Internet Protocol (IP) • by exploiting a protocol extension [38]. DStar [110] and

Aeolus [10] Also apply control? Ow of information in distributed systems. It is also common that tracing is e? Ectué in several levels at once, for example within the Java virtual machine and in the operating system, through cooperation mechanisms. TaintDroid and implements several trace levels within and between process and relies on the kernel to ensure the persistence of his plotting greeting [28]. Similarly, the mechanical scribing ism of greeting Panorama is implemented in hardware but rebuilt Panorama semantics for the core to provide more context to greeting [108]. In? N, DroidScope malware scan by rotating them in a virtual machine and its analyzer shades also work in hardware, each instruction of the emulated processor participating in the spread. He rebuilt the semantic view of both the core (Android, in this case) and the Dalvik virtual machine (rotating Android apps) to make sense of the greeting products studied by the software. He can trace the flow? Between Java objects for Android Apps [106].

2.2.2 Control greeting decentralized information

The control? Ow decentralized or *Decentralized Information Flow Control (DIFC)*•, allows each user of the system to choose himself the policy applied to data belonging to it, that is to say the data it has injected into the system. The DIFC was created by Myers Liskov [65 , 66]. Decentralization is e? Ective and easy in a discretionary access control system, since each? Le or process belongs to a user and the user is able to grant read access or write to other users . The system does *that applying* the policy chosen by the user (which is however limited to their own containers, users do not control what happens to their information once in the container of another user). Conversely, in the mandatory access control systems, the super administrator puts strong restrictions on possible permissions for each object? No guarantee strong policies. In addition, many systems such as Bell-LaPadula hypothesize the *stability,* that is to say that the containers are not supposed to change the sensitivity level. Furthermore, the declassified? Cation (lower the level of sensitivity of a given after audited) or conversely approval (increase the level of integrity of a given after veri? Ed) operations are not designed to be simple in the system because the slightest error can lead to information leakage or corruption and reduce the usefulness of nil safety mechanism. so users have no control over their own data. In a simplistic system of control? Ow of information, the situation is similar. Users must rely on the administrator to apply the correct labels on their own? Les, and especially for the modi? Er, and this is not without di? Culty.

In one DIFC , the notion of *owner* is very important because it is a delegation of administrative rights. In terms of control? Ow of information, it means? E the right to change? Er or even remove the part of the label contributed by the user at a given information container. For example, imagine a very abstract system with three users: Alice, Bob and Charlie. Alice wants to send a love letter to Bob without it (a bit boastful) can show it to Charlie. A simple way to do this is modi? Er the message label to incorporate some mark and then ask the system to apply a single policy stating "Bob can read the containers with this brand," "Charlie can DO

NOT read containers with this brand. " In this way, Bob, by downloading the message and saving it in a? Le will sound? Le scored. He will be able to read, but not to show it to Charlie. Later, if Alice wishes, she can lift this restriction by simply updating the policy, or by removing the mark of? Le.

In this section, we've established the founding principles of control? Ow of information, the di? Erence in relation to access control and modalities of its implementation, but we have not discussed how to the implement in a practical operating system. This thesis focuses in particular on the implementation and veri? Cation of the implementation, a monitor? Ow of information in the Linux kernel. In the following section, we will therefore study the history of greeting information control implementations and the principles and inventions that have guided the development of this field of research, before we focus on implementations dedicated to Linux .

2.3 Implementations of control? Ow of informa- tion in operating systems hue propa- gation

In this section, we are particularly interested in implemen- of greeting information control tations by hue spread across operating systems. This category includes both whole operating systems [56], The nuclei of operating systems [100 , 109] Safety components for these nuclei [17 , 33 , 51 , 67 , 76], Monitors implemented as programs monitoring the greeting from the outside of all or part of the process of core [28 , 51] Or even *Application Programming Interfaces (APIs)* · allowing the development of distributed applications controlling the greeting between the processes of these applications [10]. The only criterion is that the containers are considered information on the granularity of the operating system (process? Les, Inter-Process Communication (IPC) · miscellaneous, etc.) and that ux? are characterized by calls to the operating system process.

Note that there is not really waterproof barriers between di? Erent levels of application (hardware, language, system, etc.) control vows. In e? And the work of Denning [20 , 21] And Myers Liskov [64 , 66] Have very general models which are then applied to the special case of programming languages but detailed operating systems model lowest reference it nevertheless [51 , 100 , 109]. The distinctions we have introduced between "control? Ow of information," "mandatory access control" and "hue spread" are not present in all articles and works. We recall here that the di? Erence fundamental that is interesting in this thesis is that the control? Ow of information required to maintain a state of the flow? Past that occurred in the system. If the state takes the form of a label attached to objects (including system processes) and evolving progressively as greeting happen, we'll deem it comes to color spread .

2.3.1 Implementations of customized operating systems

IX

The first prototypes of operating systems centered around the control flow of information have been custom built for this very purpose, a? N demonstrate the implementability and benefits of this type of solution. They reuse portions of pre-existing systems, deep rewritten to incorporate the mechanism of propagation hues. The first operating system to be implemented control flow information, to our knowledge, is IX by McIlroy and Reeds in 1992 [56]. This system is a variant of the UNIX operating system that supports strong security policy at several levels in the tradition of the recommendations of the *orange Book*

of DoD [53]. McIlroy and Reeds describe their system as applying the mandatory access control. In fact, according to our disappointed definitions given earlier it is good control? Ow of information. In e? And, several factors distinguish clearly IX Bell-LaPadula model or even *High Water Mark.* First, IX abandon the subject-object duality and no longer considers as information containers involved in greeting (or in their words " *qui entre data occasionally seats? ows* "). Second, the authors make the observation that unlike the access control where only the openings of files? Need to be controlled, control of vows requires to apply the policy to each read operation and write because labels can be modi? ed to each greeting.

McIlroy and Reeds also explain the problems related to the fact that control of greeting itself creates hidden communication channels. The example given by the authors is that of a process with a low level of authorization wanting to get data from a working process, but much higher level. SomeMethod simple to implement and relatively? Ble is up process of writing or not in a? Le low sensitivity level initially. If he writes, this automatically increases the sensitivity of the? Le. Down process can easily observe the level of shit trying to read: if the read fails is that the sensitivity was high. In this way, the process above can therefore communicate an information bit at the bottom process. The operation can be "industrialized" by repeating at regular intervals (and replacing the? Le to bring the low) is? N to transmit any data one bit at a time. In? N, the article also discusses the problem of *explosion of colors,* which is the name given to the phenomenon doing that in a system applying control? ow of information, to? the time, the containers tend to acquire labels increasingly restrictive, and that the possibilities of greeting to information thus become increasingly restricted. The problem is similar to that of a program that always allocate more memory without ever releasing it. This endangers the availability of the system, which is a security objective.

Asbestos

Asbestos is another example of a new operating system designed for data protection and insulation services. Its creators, VanDeBogart et al., Preach to explore new opportunities in the context not re- hugged a new operating system [100]. One of the new ideas as a major achievement is the decentralization of greeting information policies. In e? And, Asbestos offers not only a comprehensive policy to provide traditional collateral such as the DoD but also the opportunity for each application de? ne its policy of isolation and protection vis-à-vis data

other system processes. In? N, Asbestos offers, for the first time to our knowledge, a way to fight against the explosion of colors by a mechanism known in the context of the database as the *poly-instantiating [* 55]. Reference is with pro? T in the article "Solutions to the Problem Polyinstanciation" [43] For a detailed presentation of this mechanism in its original context. The poly instantiation is to separate states or sessions that a server maintains with di? Erent clients. Usually, the only goal is that of insulation, a? N ensure con? Dential data of each vis-à-vis other processes. In a sys- tem to spread hues, this mechanism also prevents contamination shades that prevent the server to apply to more than one client. The alternative could obviously be dynamic server launching a di? Erent for each arrival of a new customer but would have a significant cost in terms of resources. The poly-instantiation, if supported by the operating system that ensures proper isolation between di? Erent sessions of the same process,

Asbestos labels are organized around the concept of tags, which are identi? Ants associated with certain classes of content. Tags are allocated by the utilisa- tors of the system and each administered by its allocator. Labels are just disappointed ned as a set of tags, from di? Erent users. When a flow of information occurs from one container to another, the tags for the source are added to those of the destination. The allocation of a tag automatically confer the privilege to add this tag to label any container owned by the user and remove the label any container system (of course, if any user had the right to mark any container with its tag, it could prevent the legitimate owner to access it). This model thus gives each user the right to de? Ne a policy on its own data, while being constrained by political de? Ned by other users on the tags that are not theirs. The system is the guarantor arbiter of good policy implementation. Asbestos is an implementation of DIFC in an operating system [65].

The "new primitives" qu'Asbestos promises boil down to one central element: message passing. The messages are treated as objects in their own system and therefore obey the rules outlined above. To prevent that these rules create hidden channels, Asbestos makes these interfaces *right? ables.* If aMessage sending fails because of knowledge? Cient permissions and the revealing expose information on the labels of *recipient,* then the error is not reported to the transmitter. ? This of course causes problems because it is difficult for an application developer can not rely on channels not reliable, including locally - it is more usual to only consider communications from machine to machine not reliable?. To understand what hidden channel is avoided by this mechanism, we can consider two processes, a process *H* can read the marked tag content *t*

but not having the opportunity to declassified? er and process *The* that can not read. Suppose the process *The* wants to read? shit marked *t* with the complicity of the process *H. H* may allocate a new tag *t* ·then create a communication port that mark or not high integrity with the new tag. Then, *The* trying to send a message to that port. If the port has not been marked with *t* · the message *The* may happen, but if the port has been marked as *The* does not have the known level of integrity? health, *The* has no right to send this message. If the error is related to *L,* depending on whether the transmission fails or not, *The* known whether *H* tagged or not its port, which allows *H* passively send one bit of information to *The* free chat

with him. *H* can thus send *The* the? le that he has no right to read, bit by bit.

HiSTAR

HiSTAR is an operating system directly inspired Asbestos but focusing on inventions di? erent [109]. HiSTAR reuses tags Asbestos but requires all ow? Are explicitly requested. That is to say that unlike Asbestos using a model *label? oats,* where are the? ow of information that automatically cause a change of labels of the processes involved, HiSTAR uses a model *explicit elevation* labels. The process eager to be the destination of a greeting information must first raise his label to a level su? Cient to overcome the label of the source of this greeting. This bias closes a hidden channel. Like Asbestos, HiSTAR applies a control greeting decentralized information, the process that creates a tag being free to add and remove all objects belonging to him. Through this mechanism, HiSTAR can do without superuser. However, since the labels model is very abstract and general, administration of such a system is complicated. HiSTAR gives several examples of policies to be applied to specific software but the composition of the policies applied to di? Erent components within the same system is not discussed. It seems di? Cult to demonstrate in particular the consistency of a security policy, especially considering the dice? its maintenance over time.

Another point of similarity is the fact Asbestos qu'HiStar disappointed nit primitive communications news, with some similarities to those of some UNIX systems but remaining unique in their operation. First, the concept of generalized HiSTAR *repertoire* traditional systems in that of *container.* A container is a kind of directory referencing the existence of other objects may themselves be containers. The containers bear a label and it allows to hide the existence of certain objects to authorized non process. Allowed processes can, themselves, check the container to know the labels of objects that are referenced there and know how they should raise their label for access. The segments and address spaces are also managed by HiSTAR objects. Every process has a certain address space which represents its virtual memory consisting of a plurality of segments, each with a label. This system allows a *thread* written in memory of another safely. However, it is not clear that this mechanism can not be bypassed. In e? And, if a process has two segments read-write, if one of them has a label con? Dentiality stronger than the other, the process can still write the first into the second without system call (the reads or writes to memory are entirely in user space, since the segments are loaded in memory), and therefore without qu'HiStar can see this declassified? cation illegal. This point is not discussed in the article "Making Explicit Information Flow in HiSTAR" [109]. In? N, HiSTAR disappointed nes *doors* which are callable functions in an address space by a *thread* even if he does not have this address space, provided he has knowledge through a container. When a process passes a gate - which amounts to e ectuer a function call from another process?

- it may delegate some of its labels to it *until the return of the function.* This mechanism allows for example to implement a declassified? Cation controlled.

2.3.2 Implementations under existing operating systems

Flume

Flume is a system quite di? erent from those presented so far. Unlike Asbestos and HiSTAR, Flume is not intended to protect the entire system but solely some applications that "encapsulates". In e? And Flume is not implemented as a security module into the kernel but as a performance monitor userspace, turning in a Linux or OpenBSD system. Linux, Flume, however, requires a few additions to the kernel in the form of a Linux security module (see 2.4.2). Monitored applications also require an e? Ort to carry light turn inside Flume. Nevertheless, this e? Ort is much lighter than that consisting in rewriting applications to their bene? T from the new primitives of Asbestos and HiSTAR systems. In particular, Flume keeps CPI Linux (pipes and networks in particular sockets). Flume emphasizes the simplicity of designing security policies by structuring labels in several distinct parts of di erent parts respectively manage the con dentiality, integrity and declassi privileges cation and approval???. Flume presents n the main ant compromise justify cutting this section "custom systems" on one hand and "integrated systems to Linux," on the other:? A bespoke system as Asbestos has a range of code certi? er smaller and better control on the hidden channels, in contrast, an integrated Linux security mechanism is more likely to be adopted and e? ectively used because it bene? ts of Linux development dynamics and applications already supported by this kernel.

The Flume label model that inherits the notion of Asbestos tag as identi? Opaque ant of a certain class of information. Flume is a DIFC The process can therefore freely allocate and administer tags tags they allocate themselves. The objects monitored by Flume has two labels: one for the con dential, one for integrity?. Tags are separated by con tags? Tags Confidentiality and integrity. A label is a set of tags and intuitive semantics is very simple. If a? Le is marked with a tag con? Dentiality *c* then only the process with that tag can read it. Similarly, if a? Le is marked with the tag of integrity *i* then only the process with that tag can write to it. Each tag *t*

Flume attaches more two *capacity* $t+$ and $t-$. $t+$ represents the right to add *t* his label (or con? dentiality or integrity, according *t)* while $t-$ represents the right to withdraw *t*. These capabilities allow you to implement declassi privileges? Cation and approval. As HiSTAR, labels change a process *p*

may only be made by *p* itself and should be made explicit. The greeting does not automatically increase the labels. In addition, passive system objects (objects not executing code, that is to say, all except the process) have labels? Xed. A process of creating an object is free to choose its labels with the restriction that it must be able to write to it (which prevents a very secret process of creating a publicly readable file).

Aeolus

Aeolus [10] Is a logical result of the work Liskov and colleagues about J? Ow and his successor Jif [64 , 66] And the introduction of control greeting decentralized information [65]. Aeolus is a platform for developing distributed applications with strong guarantees on vows to information and

from the application, as well as between *threads* the component which can be deployed on multiple machines. Aeolus is implemented as a Java library with which it is possible to program secure applications. Like the authors of Flume, the authors Aeolus make the determination that it is essential to provide primitive communications, managements and labels declassified? Cation / single approval for the adoption of control? Ow of information by the widest possible audience and that only the decentralization of greeting control allows su? cient flexibility. Unlike Flume, however, declassi privileges? Cation and approval are not exercised through *capacity* but by *authorities,* a concept considered more common for directors by the authors.

All Items Aeolus and all processes (which are not the operating system process but lightweight process JVM whose launch and execution are controlled by Aeolus) have two labels, one for con? dentiality, the other for the integrity and an owner. This owner is not directly a user but rather a role; which allows users to partition their activities. As Flume, labels are composed of tags that can be created by processes to categorize their information. When a process creates a tag, the owner automatically acquires *authority* this tag. The authority makes exercising the privileges declassi? Cation and approval. The authority for a tag can also be revoked and delegated to other owners. The hierarchy of the owners is a directed acyclic graph that lets you know who administers what. In addition, significant improvement over previous approaches, Aeolus for de? Ne tags groups that can be manipulated as a single entity, and simplified? Ent political expression.

Aeolus disappointed nit several communication primitives. First, the pro- cess can use Remote Procedure Call (RPC) * to call a function of a process from another. What is interesting in this mechanism is that it allows die process? Ne the conditions under which their functions can be called. In particular, it is possible to attach an owner with specific authorities? C to each remote callable function. This allows a process for example to call a method of declassified? Cation on data when it has not itself the authority. A practical case could for example be the function of authentication? Cation. Users do not want their password is disclosed to an entire application when they authenticated? Ent can audit a short and simple function check? E their password and declassified? E the result "password correct "or" incorrect password ". *password.* Thereafter, they knew? T always send their password in a message tagged with *password* to be assured that it is impossible for the application to do anything for their password except call the function audited. Aeolus also provides a shared memory mechanism between processes based on the same rules, as well as a system? Files for persistent labels on files?. In? N, Aeolus allows applications to communicate with the outside through socket networks or other mechanisms provided by the Java language. When receiving an external data, the process "at the border" should have a blank label and integrity in case of issue, a label designed dential empty, a? To avoid any vows not allowed to circumvent the protections of Aeolus.

Implementations 2.3.3 in the Linux kernel

There are several examples of security solutions implementations using more or less complete control? Ow of information within the marketed operating systems kernel. Most target the Linux kernel. This includes since 2001 an interface to implement security modules called

Linux Security Modules [104 , 105]. The use and operation of LSM are detailed in section 2.4.2 .

Laminar

The authors of *Laminar,* . Roy et al, make the determination of complementarity between controllers flow of information to the programming language level and at the operating system level: while the language level used to trace stream at a scale not very? , the operating system level can trace the greeting in the entire system [76]. Laminar proposes a mechanism implemented in both the Linux kernel and the Java virtual machine? N to get the best of both tracing levels. Unlike most previous approaches, Laminar not trying to de? Ne new communication channels but rather to reuse existing abstractions of Linux, legacy UNIX.

Laminar mainly inherits its model labels Flume. ow of information containers are *threads* , which are the smallest unit schedulable by the kernel, and? le in the broad sense, that is to say all the objects manipulated via a descriptor? le (see Chapter 3 for a detailed discussion of these concepts). As in Flume, labels objects have two sets of tags, one for con? Dentiality, the other for integrity. Logically, a tag is used exclusively or for con? Dentiality or for integrity. All the parts of all tags naturally form a lattice and objects containing no label are characterized by two empty sets. Threads additionally feature a set of capabilities. Since a tag *t, t +* is the ability to add this tag to their label (in the case of a tag con? dentiality, this gives them the right to access the content by tinted *t;* in the case of integrity, this corresponds to the approval privilege) and *t -* is the ability to remove that tag from their label (in the case of a tag integrity, it gives them the right to read from sources less reliable;? in the case of con dential, this corresponds to privilege declassi? cation).

Laminar can enforce all the traditional policies proposed by the previous models but imposes limitations on the de nition of ux??? A stream is an information moving from one object to another, of which at least is a *thread* .
In addition, following the example of HiSTAR the *threads* must explicitly change their labels properly before a greeting to take place. The vows do not change automatically.

In addition to applying control? Ow of information across the system, La- Minar bene? Ts from more granularity? Does for Java programs through its implementation in the Java virtual machine. The Java part of Laminar may ra dinner analysis controlling? Ow of information between *threads* Java programs and objects (as defined data structures) they allocate dynamically. The Java part of Laminar does not allow manipulation of tags continuously. In contrast, areas where tags can be added or removed are restricted to *security zones* that must be de? ned by the programmer. Each area carries labels and capabilities that become those temporarily *thread* who between them. A *thread*

can access data with a label if it is inside a security zone. According designers Laminar, Roy et al., This bias reduces the amount of code it is necessary to audit to ensure good safety properties of software. Nevertheless, there are some flaws in this approach. First, it is clear that Laminar can not be used as such to protect existing applications. In e? And Laminar adds several system calls for manipulating tags. This is a modi? Cation of deep core, which complicates its maintainability. Furthermore, this means? E that the applications can not be laid directly on a Laminar system. Moreover, although the model of the Java part of Laminar is very powerful, again,

blare

blare is a greeting information monitor prototype originally developed by Zimmermann [113 , 114] Then Hiet [40] And Hauser [38]. Blare differs from other approaches presented here in several respects. First, the initial objective of Blare is not, like other tools, the application of a flow of information control policy but a tool *intrusion detection* based on a policy of greeting. The di? Erence minor seems a conceptual point of view since enforce policy and monitor violations seems to come back to the same. However, in practice, this has led the design and development of Blare atypical choice. First, Blare is not opposed to the normal course of system calls. It is inserted in the execution of system calls but merely to raise an alert when it detects a possible violation of the policy? Ow of information without returning error. Blare originally had a *daemon* userspace for veri? er the legality of greeting information. In the most recent versions, How- ing, Blare is fully implemented in the kernel. The implementation is based on the framework LSM . As Laminar, information containers include *threads* scheduled by the kernel and files? (including pipes and network sockets). Blare monitors more? System V messages and also considers the projections in memory and shared memory segments process. On this Der- deny the point, the implementation fails, however, as explained in section 6.1.2 . In chapter 6 We present a contribution to solving the problem so proven.

The model is also Blare radically di? Erent previous approaches. This is the model developed by Tong Viet Triem, Clark and Mé [101]. Labels are always elements of all parts of tags, organized as a lattice. However, each object has only one label and tags are not classified tag con? Dentiality and integrity tags. Each tag in fact characterized a primary source of certain information. Each object has two labels: an information label and a political label. The information label is a set of tags. It disappointed nit what information that the container is likely to contain at any time by representing the accumulation of information sources that have stained. The po- licy label is a set of tags sets. A container is in a legal state if its information label is a subset of at least one of the sets of their political label. This helps disappointed ne policies both con? Dentiality and integrity very simply. The tags that are present in any of the entire political label of an object are simply inaccessible to this object, which capture the as-

pect "con? dentiality," and some sources of information mixtures di? erent are prohibited, even though the information is individually permitted, which captures the "integrity" appearance. One can for example allow a? Le contains accounting bills of 2016 or 2017, but not both at once, giving the? Le political label {{ *bills* 2016}, { *bills* 2017}}. Propagation is naturally to each occurrence of a greeting information by changing the information label of the destination object by union information labels the source and destination objects. If the new destination information label is not legal, an alert is raised. Unlike all implemen- tations studied so far, no distinction is made between the in Blare *threads*

or processes and other objects, they all wearing a political label and the model predicts that the greeting can take place from one object to another even when neither is an *thread* . Several Linux system calls allow such a greeting to take place between two? Les, without that information is channeled through a process. Moreover, contrary to approaches such as HiSTAR and Laminar, Blare labels of note that are? Oats. Generally, Blare designers are more attached to cover the most possible channels open and have little considered the problem of hidden channels, including those created by their own tool.

The model Blare was declined in several implementations beyond the Linux kernel. Implementing built with LSM Linux kernel *vanilla*• was renamed *KBlare [* 33 , 38 - 40 , 115]. There is also a version for the Java Virtual Machine [40] named *JBlare.* Cooperation is possible between KBlare and JBlare, a? N ra dinner spread color e? Ectuée by KBlare. The similar mechanism of Laminar was developed concurrently and independently of that of Blare. In? No, more recently, a specific release? For Android system, called

AndroBlare, was developed by Andriatsimandefitra. The main di? Erence compared to KBlare is that Android systems have a CPI Preferred for communication between applications: *binder,* that is implemented in the kernel. In addition, applications are particularly well insulated and have much lower default privileges than have lambda process on a typical Linux system. The tint accuracy propagation is improved accordingly.

Weir

The greeting information control mechanism *Weir* designed and implemented by Nadkarni et al. [67], Is the latest prototype we've encountered in the literature, was published in 2016. As AndroBlare, there is a control monitor? Ow of information for Android, based on LSM . Weir uses the template labels Flume but unlike the latter, it uses labels? Oats, arguing their realism against the explicit change model labels promoted by HiSTAR. In e? And explicit change requires some way of knowing in advance what the recipient vows will take place before the permit, which is easy in the case of playback? le, much less in the receipt of a message from another process. Weir made an interesting contribution to limit the explosion of colors inevitable in a system labels? Oats. A service used by other processes such finds tainted by all these, and tint to turn all those who use it. After a certain time, it is inevitable that the vows legal becoming increasingly di? cult to achieve and the system quickly becomes unusable. To overcome this problem, Weir uses poly-instantiation. As Asbestos that maintains

separate sessions for di? erent users, Weir can deduplicate the fly processes or applications of the system? n control both memory footprint and CPU load applications and the spread of shades. A new instance of an application is launched each time the application is called from a security context (a label) di? Erent. Conversely the same application using twice to the same service will be served by the same body, which will therefore kept the communication status. This model is particularly relevant where Android applications are interconnected with each other without even knowing. On Android, if an application needs to send an email, it is the system consults the list of available applications that can provide this service and asks the user to choose from di? erent applications available. A single application is therefore likely to be sought from very di security contexts? Erent and it is important that the poly-instancing mechanism is transparent from the perspective of applications that require the service. In? N, Weir puts particular emphasis on convenience and backwards compatibility has? N can be integrated into an existing system, where older applications that do not create tags and do not use A single application is therefore likely to be sought from very di security contexts? Erent and it is important that the poly-instancing mechanism is transparent from the perspective of applications that require the service. In? N, Weir puts particular emphasis on convenience and backwards compatibility has? N can be integrated into an existing system, where older applications that do not create tags and do not use A single application is therefore likely to be sought from very di security contexts? Erent and it is important that the poly-instancing mechanism is transparent from the perspective of applications that require the service. In? N, Weir puts particular emphasis on convenience and backwards compatibility has? N can be integrated into an existing system, where older applications that do not create tags and do not use DIFC continue to operate.

2.3.4 Other uses tracing greeting information

So far were presented only? Ow control systems of informa- tion aimed to monitor system activity and warn or prevent greeting illegal. However, although it is the historical use of propagation tion of color, this is not the only and recent work has shown that the same tools could be applied in other contexts such as malware analysis. This is the approach applied by DroidScope [106] And AndroBlare [1 , 2].

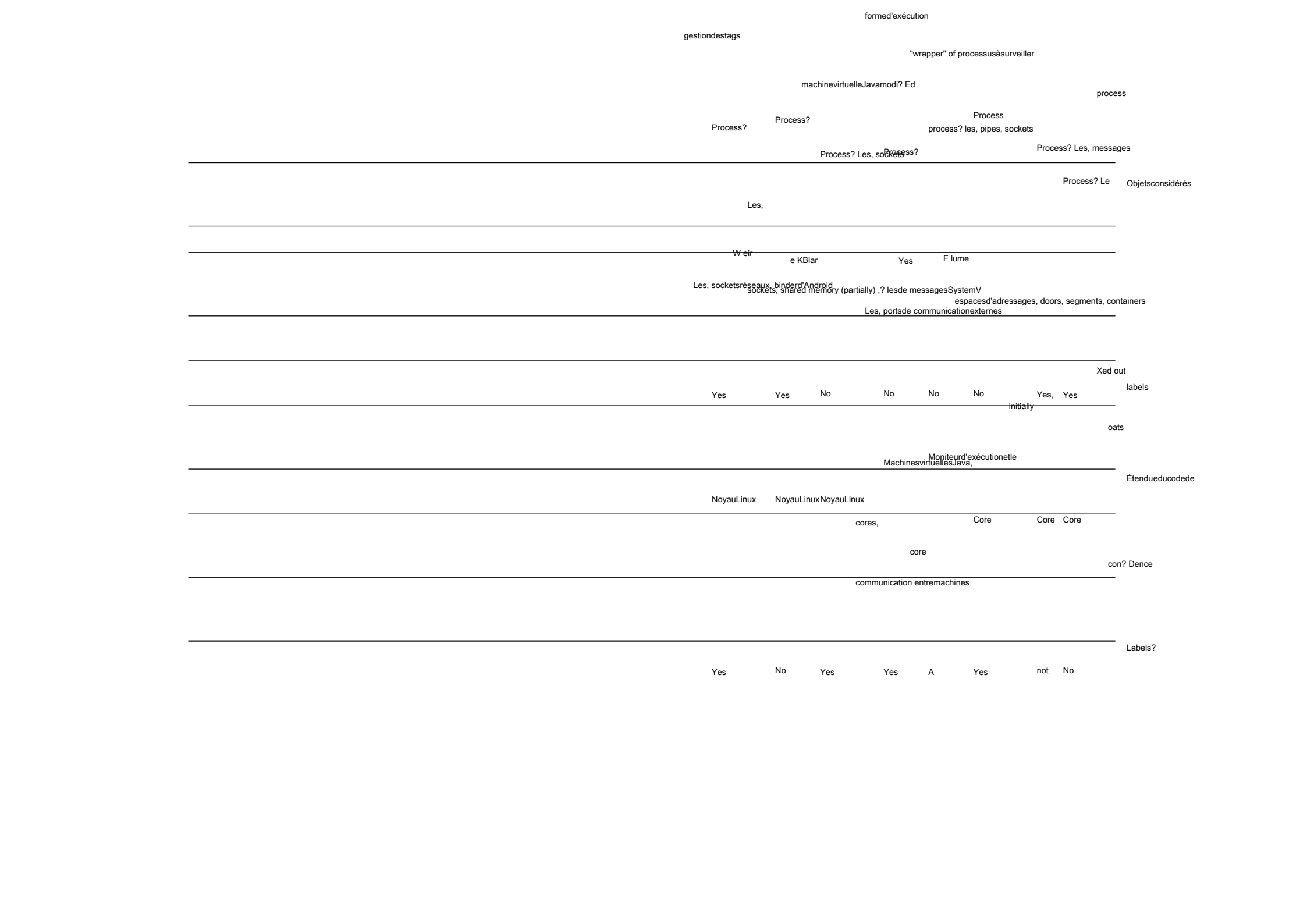
2.3.5 Conclusion on the implementations of control? Ow of information in the operating systems

The domain controllers greeting informtion is very rich and a lot of research di? Erent were devoted to the problem in the best possible implementation. In fact, the design of a controller? Ow of information involves making many compromises.

- Is it better to implement a new system or gre? Er a safety mechanism on an existing system? The first solution allows demieux control the communication channels whose processes have but the second promotes maintainability and adopting e? Ective of the proposed solution, ultimately.

- Is it better to cover as much of the existing channels, or just restrict opportunities CPI o? ered in the process? Again, the first solution seems more secure but the second is more realistic; existing channels are used by at least one application, it will re-implement if this channel is blocked.

- Is it valid to change the semantics of CPI existing if it avoids greeting illegal? Asbestos and its descendants, typically, make no pipes? Ables by not allowing the sender to know if his message could have happened to. This closes a hidden communication channel. However, applications that depend on the specific semantics? Ed by POSIX pipes no longer work.


formed'exécution
gestiondestags
"wrapper" of processusàsurveiller
machinevirtuelleJavamodi? Ed
process
Process?
Process?
Process
process? les, pipes, sockets
Process? Les, messages
Process? Les, sockets
Process?
Process? Le
Objetsconsidérés
Les,
W eir
e KBlar
Yes
F lume
Les, socketsréseaux, binderd'Android
sockets, shared memory (partially) ,? lesde messagesSystemV
espacesd'adressages, doors, segments, containers
Les, portsde communicationexternes
Xed out
labels
Yes
Yes
No
No
No
No
Yes,
Yes
initially
oats
Moniteurd'exécutionetle
MachinesvirtuellesJava,
Étendueducodede
NoyauLinux
NoyauLinux
NoyauLinux
cores,
Core
Core
Core
core
con? Dence
communication entremachines
Labels?
Yes
No
Yes
Yes
A
Yes
not
No

- What extent can we accept code as con? Dence? In any operating system, even those equipped with a safety mechanism, it is inevitable that some of the code must be considered con dence: the code that implements the security mechanism or may supplant.

- Is it beneficial? That deporting the controller code of greeting information to user space? The code in user space seem less isolated than the kernel code, though he toured with fewer privileges, which is a good thing to reduce the impact of vulnerabilities in its code. We summarize the characteristics of di? Erent controllers? Ow of information presented in this section in the table 2.1 . Check the greeting information means being able to *detect* all ow? a? n to be able to have a view of the state of the system in line with reality. As part of the operating systems, intercepting system calls has emerged as the natural and ideal candidate for the detection of? Ow of information. In the next section, we show the user space concepts, core and system calls, and we study the reasons, advantages and disadvantages of intercepting system calls for the implementation of security mechanisms throughout the system exploitation.

2.4 Interposition in system calls

Unlike a programming language or a microprocessor, there is not really die? Nal list of "instructions" of an operating system. In the remainder of this speech, one considers an abstract operating system consists of a
core, driving hardware and orchestrating the life of the system, and a *tor space utilisa-* including all other programs, they actually belong to users or whether portions of the operating system as programs performing administrative tasks in the background. The kernel and userspace programs live lives very di? Erent. Table 2.2 illustrates the main di? erence between the core and the utilisa- tor space programs. There is a natural border between kernel and user space constituted by the interface *system calls* - a term that actually would have been better translated as
system calls. In e? And no kernel function can be called from user space to share system calls. If we assume that all operations causing greeting information are implemented in the kernel, the kernel can therefore be considered as a machine to make vows, and all the system calls like his game instructions. To control the flow?, It is known? Cient to a safety mechanism to mediate between user space and kernel in the system calls.

2.4.1 Risks to the interposition of secu- rity checks in the system calls

Garfinkel studied in "Traps and Pitfalls" [31] The risks to a safety mechanism monitoring user process activity to the system call border. It classi? E into five major categories.

Desynchronization between the state maintained by the mechanism and kernel state
Sometimes a security mechanism needs to maintain a condition for taking

Table 2.2 - Main di erence between the kernel and the processes running in kernel space user?

	Process in user space
Is responsible for the hardware and launched at system startup.	Is loaded by the kernel and is generally ment launched at the request of the user.
Do not stop, the kernel unexpected stop making the system unusable.	Can stop at the? N of his task or be interrupted by the user or over- comes an error.
A maximum privileges on the mate- riel.	Can access directly aucunmatériel, all communications should be through the core.
Has access to physical memory.	Has access to its own virtual memory, partitioned virtual memory of other processes.
Can execute all instructions of the processor.	Is limited to a certain subset of instruc- tions.
May perform any action in the system, transfer of informa- tion from one process to another, write in a? Le, etc.	Must request privilé- giées operations core, e? A ectuant *system call.*
Is very sensitive to bugs, the slightest mistake can compromise the security of the entire system and any born Don- stored there.	May jeopardize yourself and the data and programs from the same user.

future security decisions. For example, a controller? Ow of information may need to remember that a process share a memory area with another process to remember that there is a greeting of information between the two that will last at -delà calling system used to implement this shared memory. If the safety mechanism fails to keep updated the list of shared memory, it will take incorrect security decisions. Moreover, these errors tend to accumulate over time, unless it is possible to "resynchronize" with the kernel state, which is still authority.

incorrect replication of a kernel feature As the security mechanism must make decisions involving the arguments passed as parameter of system calls, it must sometimes apply the same changes as the kernel itself. In particular, when argument is a path, it is necessary to calculate the canonical form. This algorithm is notoriously complicated, among others, for historical reasons dating back to UNIX. If security lemécanisme must replicate rather than reuse the kernel implementation, it runs the risk of misinterpreting the arguments of the system call and therefore make wrong security decisions. Even more insidious - it is a point that is not directly addressed by Garfinkel, even if the implementation of the security mechanism is more correct than the

core (the point of view of POSIX standards or *Single UNIX Speci? Cation* for example), it is still that of the core authoritative because it was he who, at? n, will interpret the arguments and e? ectuera the sensitive operation in the system call.

Competitive conditions in the arguments of system calls This pro lem occurs only in the case of security mechanisms implemented in user space. In this case, it is as follows:

? 1. secure e ectue process a system call, the safety mechanism is noti ed immediately;

2. the mechanism stops the execution of the process to check? Er the system call, its arguments, etc. ;

3. If everything is correct, the system call is permitted, the process execution resumes;

4. process e? Ectue its system call.

The problem with this sequence of actions is that it is possible that the arguments of the system call changes between the time they are checked? Ed by the safety mechanism and when the kernel executes the system call. This may be the case for example if one of the arguments is a string stored in shared memory. If the security mechanism is implemented in the kernel However, the problem does not arise because the kernel always starts the system calls by copying userspace arguments to its own memory. This ensures that the arguments that are veri? Ed are those who will serve during the rest of the execution of the system call.

Partial protection of a resource If certain kernel data structures can be modi? Ed by means that are not covered by the safety mechanism, then the protections of the latter may be bypassed. A classic case presented by Garfinkel [31] Is the descriptors? Les that can be exchanged between processes via a message on a UNIX socket type. This allows a process to access a? Le opened without opening itself. If a safety mechanism seeks to control the openings of? Le but only controls the system call open and not the system calls the family sendmsg that transfer descriptors? les, they can not but fail to apply their protection. Linux must also mé? Er multiplexer system calls

ioctl and fcntl that provide access to particular functions of files and devices?. The possibilities o? Ered by these calls are highly variable because they depend on the equipment and the real operation that e? Ectuent is speci? Ed by the arguments.

Bad behavior of applications due to the interception of sys- tem calls When the security mechanisms intercept system calls, they must be able to return an appropriate error code to the process if the permissions of the latter are blind? Cient. It is important to return a code documented by the original system call because the process may have an unexpected behavior lies if it receives an error code which is not prepared. On the other hand, if the restricted security mechanism all the system calls available to a process, it must take care not to prevent system calls for *reduce*

privileges. Otherwise, it is likely that the process will ignore the error and continue still to run with elevated privileges too.

Although Garfinkel was not considered particularly if monitors flow of information, observations apply perfectly to this case, particularly the first point concerning the synchronization between the state of the? Ow control mechanism and the core.

2.4.2 The system *Linux Security Modules*

the framework *Linux SecurityModules (* LSM) [105] Is as its name suggests an interface to implement extra security modules of default existing discretionary access control mechanism for the Linux kernel. It was introduced in
2001. It has two aspects. On the one hand, additional attributes have been added to kernel data structures to which access must be restricted. These attributes are not used by the standard kernel code and security modules are free to use them to store their state, and in the case of shades of propagation mechanisms, labels of containers corresponding information structures. In addition, special features have been added and the locals call them at strategic locations of the kernel code, before the operations characterized as sensitive from the viewpoint of security by the maintainers of the kernel. These operations can be eg reading a? Le or sending a data on the network. These special functions, called hooks (*hooks* in English) in the terminology LSM can be rede? ned by a security module to implement additional controls and possibly return an error code to em- fish realizing the sensitive operation. We detail further design LSM in chapter 5 . the framework LSM thus does not *strictly speaking* an interface for inter-system call reception because in reality the hooks are implemented deeply

in system calls. There are several clear advantages to this: first, it helps to factor hooks into system calls that manipulate the same internal kernel data structures, and secondly, it allows security modules pro? ter pre arguments of the system call by the kernel. For example, the memory addresses or descriptors? Invalid files are detected before the interposition, which facilitates the work of developers of the module. Smart conceptual tion of LSM also helps cover some of the risks described in the section 2.4.1 : There is no risk of competitive conditions on the arguments of system calls, and the like LSM are part of the design of the calls, error values they can return are documented, avoiding any nasty surprises. The modules bene? T from all programmable core mation interfaces, and can see the kernel state, which can limit the synchronization problem. The partial protection problem of resources is remote meanwhile modules to the framework LSM . In e? And these are the only places where the security mechanism can not only act (that is to say prevent incorrect operation) but also more simply *watch* the system state.

It is therefore understandable that the position of the hooks in the code is critical to ensure good security guarantees. If a hook lack before a sensitive operation, then the safety mechanism will be unable to detect, and even more so prevented. Conversely, too much would harm hooks kernel performance and render very complex and di? Cult to the maintenance of security mechanisms. The proper position of the hooks LSM can be veri ed in several ways: by testing, static analysis, dynamic, the *Model checking,* etc. In the next section we detail the state of the art regarding the analysis of the kernel code, to give the general framework

in which fit the specific approaches? cally dedicated to the analysis of the position of the hooks LSM .

2.5 Linux kernel code analysis

Analyzes of programs aim veri? Cation of certain properties expected of them. Conventionally, two categories of analyzes, with their advantages and shortcomings.

2.5.1 Static Analysis

Static analysis check? Ent properties on the source code of the program, according to the declared semantics of the programming language, without the need to compile the program into machine language, much less run. The advantages of this type of analysis are as follows.

Perfect cover all code
: All code is included, even the rarely called functions can be analyzed.

Independence vis-à-vis execution platforms
: Some conditions of competition may be di? Cult to trigger during a performance. Static analysis is also abstract from such problems.

Designed ment and reproducibility
: Unlike a dynamic analysis that requires execution and therefore can be in? Uenced by the runtime environment, static analysis can be applied in complete isolation.

2.5.2 Dynamic analyzes

Dynamic analysis consists of a controlled delivery of the program, where data on the effective respect of certain invariants, the use of system resources on the execution time, the IO, the system calls, etc. . can be collected. This type of analysis has additional benefits as static analysis.

Independence vis-à-vis the language, compiler, etc.
: Few popular programming languages have a formal semantics, most are content standards expressed in natural language. Therefore, many programs are under-speci? Ed. In addition, the compilers being themselves in software, they are not free from errors, nor the hardware platforms. Therefore, it may be di? Erence between what expresses the code of a program and its execution e? Ective. Dynamic analyzes are immune against such problems.

Ability to check? St of invéri properties? Statically reliable
: Under Rice's theorem any nontrivial property of a program is statically undecidable in the general case [74 , Corollary B] therefore static analyzes are forced to make approximations. Dynamic analyzes are able to veri? Er much more calculable properties when running.

2.5.3 Tools dedicated to the analysis

In this section, we describe the tools that have been used for the check? Ca- tion of the Linux kernel, and in particular the integrated tools in the toolchain and kernel testing. For a state of the most complete art of all the theory and tools related to static or dynamic analysis of code such as the Linux kernel, we will consult with pro? T doctoral thesis Slabý [83].

static approaches

Many types of tools were used to veri? Cation of the Linux kernel. The most basic method and faster to find errors is to look for *reasons* in the code known to be symptomatic of certain classes of problems. For example, it can detect the omission of a deallocating memory in a function or a direct comparison between a pointer and an integer. Originally, these reasons research were made by simple regular expressions but of course the grammar of C language is not regular, this method is neither right nor safe: there is no guarantee of all problems, nor do find that problems in the output of the tool. In response to this problem has been developed *Ladybug [* 69]. Ladybird is a patch engine *semantic,*

that is to say, instead of working on simple regular expressions, he understands the grammar of C and can be based on some knowledge of the language semantics such as the commutative property of addition and multiplication, type conversions, etc. This allows it to recognize patterns and apply complex transformations. Beetle is integrated into the Linux kernel sources since 2010 [97 , 74425eee71eb44c9f370bd922f72282b69bb0eab commit].

Linus Torvalds, creator and maintainer-in-chief of Linux, has also developed a semantic analyzer for C called Sparse [7 , 98]. Sparse The initial goal is not to build a compiler but only an analyzer capable of former milking characteristics of the code discussed or even propose visualizations (as graphs? Ow control, typically) . Some veri? Cation static are also facilitated by this tool. For example, annotating functions with locks information they take and they release, it is possible to Veri er that at any point of a function, the number of occupied locks is independent of path followed to get to this point (which could be a sign that along some execution paths, all locks taken are not released properly) [96].

MECA [107] and *CQUAL [* 30] Were used to veri? Er the proper use of dotted tors from user space into the kernel [47 , 107]. In e? And, for safety reasons, it is critical that the core is completely isolated from user processes. Therefore, care must be taken in the core, not dereference pointer provided by user processes (such as system call argument) without veri? Er to point e? Ective worms the memory to which the process has legitimate access. Special functions copy_from_user, copy_to_user,

etc. are available to properly dereference these pointers. The work of Yang et al. [107] One hand and Johnson and Wagner [47] On the other hand are designed to ensure that no user-space pointer is dereferenced other than those functions. MECA is based on the spread of shades: the pointers from the process are automatically marked with tinted entry systems and only calls the functions mentioned above can remove the tint. Dereference a pointer tinted is reported as an error. The CQUAL approach, for its part,

is based on type theory. Each type pointer ra born into two categories: user pointer type and type core pointer. Conversions are only permitted by the functions given above. ? The main di erence between these two approaches is that of CQUAL is correct: all potentially mishandling pointers are detected, the prices of many false positives (that is to say, reports of errors that do are not). Conversely, MECA does not claim to correction, it is possible to deceive with some complicated code and therefore mistakes can escape it, but it brings in a lot fewer false positives.

In? N, *BLAST* [4], Another static analyzer, was used to detect errors in the capture and release locks, which can lead to deadlock in particular [61]. This approach has proven conclusively, but a bit disappointing. In e? And at the time of the work of Mühlberg and Lüttgen [61], Support for pointers BLAST was too little developed to allow relevant analyzes. This led project developers *Linux Driver Veri? Cation* to produce a new module to improve this aspect of BLAST [81].

An overall limit of these approaches is that Linux is not written in C strictly comply with the standard, but in a dialect of C disappointed by either the compiler later
GNU Compilers Collection [89] (CCG) [1]. GCC is the Linux kernel reference compiler, and for a long time was the only compiler capable of compiling the kernel in- tégralité. Some extensions to C provided by GCC are explicitly permitted in the core [77]. In addition, the norm of C is written in natural language and contains ambiguities and leaves some details "de? Ned by the implementation" (meaning, compiler and standard library). The same code can therefore correspond to several executable whose behavior di? Erent. Therefore, a program that depends on another syntactic and semantic analysis than GCC runs the risk of not interpret the code in the right way and to draw false conclusions; despite its correctness proof which rests on the implicit assumption that the semantics that gives the code is the same as GCC . Since 2014, Linux kernel development, as part of *Kernel Self-Protection Project,* also integrates the gre? ons [1] at GCC in the kernel sources to improve safety [14], Copying it into the group's work *PaX* [86]. Among the gre? Ons built or proposed so far in the nucleus, one can point

KERNEXEC [12 , 24] and *structleak* [13 , 24]. The gre? We *KERNEXEC* aims to prevent the execution of code from user processes from the core. This could happen by deceiving the core and passing in an address to which to jump into the code in user space instead of the kernel, which is to allow the execution of arbitrary code with kernel privileges. The gre? We *structleak* as to oblige him that the structures marked as possibly containing userspace addresses (and therefore likely to be exposed to user processes) are initialized with known values. If this is not the case, there is a risk that an uninitialized structure reveals the previous contents of the memory allocated to it, which may include internal data core pointers revealing the organization of its memory (thus facilitating the exploitation of certain vulnerabilities), etc.

1. Perhaps better known by the English name *plugins*.

dynamic approaches

The main problem detection method in the Linux kernel remains at this hour test. Nevertheless, many approaches have been proposed for veri? Er more extensively and systematically the proper kernel behavior. A rather old approach and experiencing a resurgence in popularity now is the

fuzzing [59]. Fuzzing is to test software extensively in her pro- ning all combinations of inputs and possible events, so as to show off poorly planned behavior, or rarely borrow tees execution paths in the code. Fuzzing was purely random origin but *fuzzers* the latest utilize techniques from both static analysis and learning arti? sky? n to seek speci? cally suscep- patible inputs to be the most problematic [8 , 9]. For example, they will look to test the limits loops indexes and tables in functions to generate overflows, or they will try to trigger competitive conditions in multithreaded code if they detect a variable is shared. Fuzzing and removes the choice of using human tests is to only focus on use cases provided for in the original design of the code. Fuzzing uses fairly standard way to detect plant conditions making software [82] But can also be used to find vulnerabilities to attacks of the con? Dentiality or integrity [57]. In e? And malformed inputs can lead to dodge certain safety tests and some mistakes modes can, instead of the crash, put the software in such a state that it will be more vulnerable to other attacks.

Fuzzing has been used repeatedly to test Linux. The former haustive list of works of this branch would be off topic here but one example is the work of Sim, Kuo and Merkel [82] Or Mendonca and Neves [57]. Regarding the tools dedicated to fuzzing, they are also likely to have succeeded and are used extensively to test the new device drivers and critical interfaces such as system calls. *Trinity [* 48 , 49] Is a fuzzer for Linux system calls. It has in its knowledge base a number of notes on the system calls that allow him to choose intelligently the arguments to pass to the appeals system? No trigger interesting errors that daunting error code EINVAL (invalid parameter). In particular, Trinity has a notion of "type" of Linux objects. If an argument is supposed to be a descriptor? Le, Trinity will first open? Les in di? Erent kinds (pseudo-? Les, directories, pipes, network sockets, etc.) to recover descriptors and try and to test ways of carry-execution beyond the simple test of validity of descriptor. Trinity led to the discovery of numerous bugs [2], including out of the system call interface, especially in the network stack. More recent, *syzkaller [* 22] Uses the same approach as Trinity using the kernel extracted information for fuzzer system calls and is more able to examine the code to determine which execution paths he has covered. This allows it to build its appropriately inputs to continue fuzzing. This method requires instrumenting the kernel code at compile time to provide evidence of the paths followed by executions. The fuzzer syzkaller also emphasizes the reproducibility of crashes, a? N to build regression test [3].

2. See http://codemonkey.org.uk/projects/trinity/bugs-found.php for a list.

3. A regression test is a test e? Ectué to ensure that a problem missing a version apparent Raisse (or appears), not in a future release.

Many instrumentation are possible with modern versions of the compiler GCC . *KASAN (Kernel Address Sanitizer)* is a Linux kernel compile option to produce a core test image wherein for each eight-byte group, a byte is reserved to track the use made of the memory and in particular monitor that objects allocated n ' is used before being initialized or after being released [25]. Other tools exist in the same style as *UBSAN (Unde? Ned Behavior Sanitizer) [* 78] That monitors the use of operations described as having an independent behavior? Or the semantics of the language C as dereference a null pointer, the overflow of a signed integer, etc. These tools are particularly useful in conjunction with fuzzers to detect entries that push the core to adopt specific bad behavior? Ed.

Other projects and programs exist to improve the Linux kernel code and hunt programming errors by systematic testing methods. Beyer and Petrenko militate for example a consolidated implementation of all the prototypes of the state of the art in veri? Cation software, based especially on the *model checking.* Among the active projects on the Linux kernel testing, especially on device drivers that represent a part of the code signi? Cant and very prone to errors because of the interaction with the material include *Linux Driver Veri? Cation [* 50] *Avinux [* 71] and *DDVerify [* 103].

2.5.4 Application to the problem of positioning the LSM hooks

The works present in the literature about the veri? Cation of the right placement of the hooks LSM are due to Jaeger and several teams with whom he collaborated. Here we give an outline of these approaches and we will compare their relative merits in chapter 5 .

Veri? Cation of the correct positioning of brackets

The earliest work on the right position of the hooks LSM were presented along with the framework itself in 2002 [112]. This is a veri? Cation static, on the kernel code with C CQUAL, ensuring a hook LSM is good in every execution path before a sensitive operation. Sensitive operations are identi? Ed as readings or mo- di? Cation of certain data structures manipulated by the system calls. Veri? Cation security is "low-level" in that it controls access to internal objects in core, not user-level abstractions such as? Files, network sockets, etc. This property alone is not su? Health, quoiqu'essentielle. In another article [42], Jaeger, Edwards and Zhang propose veri? Er that the correct permissions are claimed for each sensitive operation. In e? And each hook corresponds to a specific function of the security module and in the called function, the module will not base its decision on the same elements. To simplify a concrete example, if a given process has the right to read a? Le, but not the modi? Er, it is vital that the hook corresponding to the read permission is called before the read operation and corre- hook laying in writing right before the write operation, otherwise the bad security decision is made in both cases.

automatic placement of hooks

Other approaches were then presented to automatically position the brackets, which has the advantage of facilitating the kernel maintenance while continuing to maintain the good desired security properties. In the article "Leveraging" choice "to automatic authorization hook placement" [63] Muthukuma- ran, Jaeger and Ganapathy present a general methodology for the correct and automatic placement of safety hooks in the general case (not limited to

LSM). They then validate their approach by comparing the investment selected by expert developers selected software for example and exhibit some prob- lems in them. In the article "Maintaining the correctness of the Linux security framework modules" [26] Edwards and Jaeger consider only LSM but include the dynamic aspect of the nucleus. In e? And the kernel development pace is fast, with a new release every eight to ten weeks about [4] and therefore, the veri? cation regression security properties provided by the hooks are required.

To our knowledge, there is therefore no work has focused on the position of the hooks to track greeting information. In e? And, LSM was designed primarily to implement access control mechanisms and therefore not intended for control of greeting. However, we can see that there is already tracking implementations greeting information as KBlare, Laminar and Weir based on LSM . The question of whether LSM a viable framework for the? ow control mechanism of implementation is open and current problem, and consitutes the subject of this thesis.

2.6 Conclusion

The study of literature in the field shows a great diversity not only of greeting information monitoring tools but also their objectives. Each author goes guarantees con? Dentiality and integrity di? Erent, and may not assign the same direction as the other under the same terms. While some tools are designed to cover the most open possible information channels, others are designed to address the problem of hidden channels. Some tools even care about the explosion of tags. We note however that the majority of these studies have focused on political issues of vows, or of application programming model. Our review of the state of the art has shown that the monitoring flow in itself, that is to say only the discovery of greeting taking place in the system, was he appears neglected, or at least considered a secondary problem of implementation. however we know that this is not because the research was conducted on the problems of interposing appeals system and the proper construction of the framework LSM Although in other contexts as monitoring greeting information. It remains, in our view, an identification equivalent work? Cation and resolution of specific problems? That monitoring of greeting. As much as possible, formal methods should be preferred. While it is unrealistic to expect to achieve an implementation "proved correct" monitoring greeting in the Linux kernel, formal methods provide necessary guarantees in terms of code coverage and maintainability over time implementations mechanisms of security.

4. Development and kernel versions are visible on http://www.kernel.org .

chapter 3

Linux kernel information Containers

In this chapter, we describe some elements of the Linux kernel implementation has? N to specify the subject of our research and our contributions in the following chapters. We give first some general information to locate the kernel development context and then present the data structures that correspond to information containers we vaguely introduced previously. This chapter was written in large part on the basis of information works *Understanding the Linux Kernel* Bovet and Cesati [6] and *Linux Kernel Development* Love [54] As well as through the work of Corbet, Edge and Sobol, site publishers *LWN.net News from the source [* 15].

3.1 General

Linux is causing the personal project of a student, Linus Torvalds, who started to develop in 1991 after the acquisition of a computer with an Intel 80386. The system is in? Uenced parMINIX which was the subject of study Torvalds, but the design is radically di? erent. In fact, Linux is much less classical and innovative point of view of architecture that many older operating systems but focuses on performance and stability.

Linux kernel sources include, for version 4.7, about quinzemillions lines of code spread over nearly forty-three thousand? Le [1] organized into twenty-two directories, presented in table 3.1 . The kernel development is open to all the sources are freely licensed (*Gnu Public License v2)* but the integration process of the code in version o? cial kernel is rather strict. This ensures the core a relatively homogeneous style and great quality despite its size. Some *bugs* may however be long before being discovered, because of the complexity of the code [11].

A particular focus is on the core performance, stability as well as its portability. However, security is not receiving attention and is considered an objective nothing more or less than others in the eyes of

1. Statistics established using cloc [19].

Torvalds [94]. The core adapts to a wide variety of platforms and use di? Erent. You can thus count on file *arch* the specific code? that twenty-eight architectures di? erent, more special architecture *um (* for *user-mode)* that launches the Linux kernel as a regular process, mainly for testing or debugging. The Linux kernel is experiencing significant success on supercomputers [2] ,

web servers [3] and mobile phones and tablets, through the Android oper- ating system [4] . It is therefore equally suitable for environments demanding top performance at environments where size and kernel energy consumption should be minimal. The Linux kernel can also be used in "real time", that is to say, in environments where it is crucial to limit the maximum time for each core action. The handling software greeting audios or videos or those dependent security people use the real time mode.

3.2 Correspondence between abstractions of the operating system kernel and internal data structures

3.2.1 System? Virtual files

The system? Virtual files is an interface for uni? Er much of the code of true systems? Le and simplified? Er writing them. This is a set of structures and functions common to all files systems. The system? Virtual files also includes the de? Nition of system calls related to the handling of? Les, directories, etc. There are four main data structures:

struct super_block A superblock is a system? Files mounted in the tree of files?.

struct dentry The term *dentry* is short for *directory entry,* that is to say "in- directory tree. " A directory entry is a name? Le available in the tree of files?. It is a component of a path to a file.

struct inode A *inode,* for *node index,* usually results in "i-node" is a file system of? les, with its contents (or more accurately the position of content on the disk or memory) and its metadata but not his name. In e? And the same i-node may appear in several places in the system? Les (and therefore more directory entries can be referenced) or even none (the? Le is in this case called "anonymous") .

struct? the This structure represents a descriptor? Le, that is to say a refer- ence on? le opened by a process. Owning a descriptor? Le is needed for most access to the content of? Shit like reading, writing, truncation, in memory projection, etc. We describe below each data structure.

---

2. In November 2016, Linux was the operating system of four hundred ninety-eight five hundred biggest supercomputers [91].

3. 37.2% market share in March 2017 [73]

4. 68.4% market share in November 2016 [90].

Table 3.1 - Organization of the Linux kernel source

Documentation Directory API and kernel features.
arch Code varying according to the architecture for which the kernel is compiled.
block Code on block devices.
certs Code on certi? Cates embedded in the core and the signing the core and modules.
crypto Code algorithms and cryptographic primitives (the implementations in the cores can bene? t from hardware acceleration).
drivers device driver code (this is the largest directory kernel, three times bigger than arch and one to two orders of magnitude larger than the others).
firmware Images *rmware* can be integrated into the kernel source (Compatible from the perspective of the license).
fs System Code? virtual files and systems? le implemented by Linux (each in its subdirectory).
include header files of the core, the subdirectory *linux* contains in- Useful heads libraries to developers and other tools depend ing kernel.
init Code loading and initializing the common core to all architectures.
ipc Code IPCs System V as well as? The POSIX message.
kernel kernel code default directory contains particular ordonnanceur, the de? nition and handling *threads* , clocks, synchronization means, the code implementing LSM Etc.
lib Mini library not quite standard used by the kernel (which can not use a third-party implementation).
mm Code relating to memory management, especially paging.
net Code of network functions, and associated protocols (each in his repertoire).
samples Examples of use of API the nucleus for? ns documentation or demonstration.
scripts Code is not part of the kernel source but useful for de- development, to handle *patches* or to export user data from the core.
security Code modules implemented with LSM and the directory Key core.
sound Code on the recording, reproduction, manipulation his.
tools Code is not part of the kernel code but that implement tools used to interrogate the core, implement testing, etc.
usr Code generation of *initramfs* .
virt Code on virtualization (to use Linux as hyper- viewfinder or as a virtual machine).