Systems? Le

The structure struct super_block represents a system? files mounted in the tree? les. It shall include a list of all i-nodes? System files and a link to the system root. It is possible for a module LSM attaching a security structure to a system of? les, especially to control the assembly, disassembly, reassembly and options. However, none of the monitors? Ow of information studied only consider systems? Les such as information containers.

```
struct {super_block
    [...]
   unsigned long s_blocksize; / * Size of a block on the disk * /
   loff_t s_maxbytes; / * Maximum size of a? le * /
   struct file_system_type * s_type; / * Type of system? Files * /
   const struct super_operations * s_op; / * Operations System
        les: allocating an i-node, synchronization on the disc * /
    [...]
   unsigned long s_magic; / * Identi? Ant system type? Le * /
   struct dentry * s_root; / * directory entry corresponding to the root of the system? files * /

    [...]
# ifdef CONFIG_SECURITY
   void * S_security; / * Safety construction LSM * /
# endif
   const struct xattr_handler ** s_xattr; / * Handling operations extended attributes * /

    [...]
   tank s_id [32]; / * Device name with the system? files * /
   u8 s_uuid [16]; / * Identi? Single ant system of? Le * /
    [...]
   const struct dentry_operations * s_d_op; / * Operations default system directory entries? Le * /

    [...]
   struct list_head s_inodes; / * inode list system? files * /
};
```

Directory entries

Directory entries and their organization hierarchically form the tree? files that users are the usual experience. Nevertheless, these are only names, links, providing access to? Les and not a representation of them.

Directory entries have no structure LSM because it does not correspond to an abstract presented to the user process. In e? And the process essentially manipulate files? As i-nodes or via a descriptor? Le. When a system call takes a nameOf? Le or a path parameter, as a string. Type structures struct dentry

representing the directory entries are actually created on demand, when a path is given as a character string and the? file corresponding must be sought in the tree? les. If a module LSM Or even a

Monitor greeting information wants to control the creation, deletion or renaming of directory entries, he can do it according to the structure associated with the i-node of the directory and path. In e? And rename a? Le consists of the modi? Cation of the inode representing the folder containing the? Le. For the core, a folder is? File whose contents are the list of files? And subfolders stored inside.

```
struct dentry {
    [...]
  struct dentry * d_parent; / * parent directory * /
  struct qstr d_name; / * Name directory entry * /
  struct inode * d_inode; / * I-node of the directory entry * /
    [...]
  const struct dentry_operations * d_op; / * Operations of the directory entry * /

  struct super_block * d_sb; / * The system? Les in the directory entry * /
  void * D_fsdata; / * Data input specific directory? C the system
      files * /
    [...]
  struct list_head d_child; / * List of subdirectories of the parent directory * /
  struct list_head d_subdirs; / * List subdirectories of the current structure * /

    [...]
};
```

I-nodes

The i-node is the central structure of the system? Virtual files. I-node repre- sents which commonly appears to users as a file process, that is to say a sequence of bytes that can be read and possibly written by the type of? Le. The i-nodes represent the part common to all types of files: regular files, directory, tubes, network sockets, devices, special files that reading, return the output of a kernel function, etc.? The particular fields associated with certain types of? Les are stored in a separate structure, accessible from the inode. The i-node represents both the content of the? Le and its metadata such as its size, the last time he was modi? Ed, etc. This structure belonging to the system? Virtual files, a field is for the real system? files bearing the y i-node combines its specific data, including location of the content of the? le. The i-nodes are permanently stored on disk. Instances of the structure struct inode are created when the? le is used and populated from the information on the disk. The correspondence between the inode and directory entries is left free to each system? Les.

This structure has a structure LSM which is often, depending on the capabilities of the system? les, serialized with the content of the i-node and is therefore a persistent aunt reboots the system. This is the main structure on which the modules are supported LSM to take their security decisions.

```
struct inode {
  umode_t i_mode; / * Fashion and UNIX permissions of the i-node * /
    [...]
  kuid_t i_uid; / * Identi? Ant the owner * /
```

```
    kgid_t i_gid; / * Identi? Ant owner group * /
      [...]
# ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl * i_acl; / * extended permission * /
      [...]
# endif
    const struct inode_operations * i_op; / * Operations of the i-node * /
    struct super_block * i_sb; / * System? Le belongs i-node * /
    struct address_space * i_mapping; / * List of process address space in which the? Le is projected * /

# ifdef CONFIG_SECURITY
    void * I_security; / * Safety construction LSM * /
# endif
    unsigned long i_ino; / * Identi? Ant the internal inode * /
      [...]
    const unsigned int i_nlink; / * Number of directory entries associated with the i-node * /

      [...]
    loff_t i_size; / * Size of the i-node * /
    struct i_atime timespec; / * Last access timestamp * /
    struct i_mtime timespec; / * Last timestamp modi? Cation * /
    struct i_ctime timespec; / * creation timestamp * /
      [...]
    const struct file_operations * i_fop; / * Default Actions descriptors? shit this inode * /

      [...]
    union { / * particular data structure depending on the type of? file * /
        struct pipe_inode_info * i_pipe; / * Data structure specific to the tubes * /

        struct block_device * i_bdev; / * Data structure specific to block devices * /

        struct cdev * i_cdev; / * Data structure specific to the device node * /

        tank * I_link; / * Data specific to symbolic links (in fact, this is the way of the le pointed) * /

        unsigned i_dir_seq; / * Data own directories, for synchronizing access to the structure. * /

    };
      [...]
    void * I_private; / * Data specific inode? That the system? Files * /
};
```

Descriptors? Le

Each *thread* can possèder its own table of descriptors? files or share one of his *thread* parent. This reference table the? Le he opened. A descriptor? Le is obtained by the system call open. In addition to the directory entry past the opening, the descriptor? Le has a cursor indicating where e will? Ectué the next read or write in? Le (except in special cases? Sequential files, which can not be played in strict order, and? le open mode O_APPEND, where all writing 'E? ectue to? n).

There is a security structure LSM in the structure struct file but it is little used by the monitors of greeting information because represent the contents of the le, and therefore its colors, is the responsibility of the i-node? le, and not of its descriptors. In e? And if a change process? Es the? Le via its descriptor, all other processes see the modi? Cation even if they do not share the same descriptor? Les because modi? Cations relate to i -node referenced by the handle and not the descriptor itself.

```
struct file {
    [...]
   struct f_path path; / * Way? Le represented by the descriptor * /
   struct inode * f_inode; / * I-node descriptor? Le (cached) * /
   const struct file_operations * f_op; / * Transactions descriptor
        le (reading, writing, moving, etc.) * /
    [...]
   fmode_t f_mode; / * Fashion and permission to open the? Le * /
    [...]
   loff_t f_pos; / * reading / writing position in the? le * /
    [...]
   const struct cred * f_cred; / * Authorizations of the process that opened this descriptor? Le * /

    [...]
# ifdef CONFIG_SECURITY
   void * F_security; / * Safety construction LSM * /
# endif
   void * Private_data; / * Data associated with the descriptor dependent? Le underlying * /

    [...]
   struct address_space * f_mapping; / * process address spaces wherein the? le is projected * /

    [...]
};
```

3.2.2 Data Structures related to process

process and *threads*

A process is an object of the operating system whose business is to char- ger a program written in a? Le and run. Its existence is determined by the start and? N of the executed program. A process has resources from the operating system. A key task of the core is to arbitrate the use of resources between processes, namely the processor, memory, peripherals, etc. This is the kernel have the burden of assigning time quotas during which the process can e? Ectively run. When the quota is exhausted, or when the process voluntarily releases the processor - pending an input-output, typically - the core backup process state then selects another process can run, restores the state of the latter (the processor registers essentially) and? n stimulus execution. Meanwhile, the process interrupted in its execution is paused. This is called the

context switching. The core mission of assigning time quotas each process to choose the right candidate to run and to start or interrupt executions are called the *scheduling.* The scheduler is a portion of the code particularly central and critical nucleus, and also technically complex. In addition, it greatly depends on the architecture of the processor.

A process can be composed of several parallel execution sequences called *threads* . For example, a web server can handle several simultaneous queries neously delegating processing each a *thread* . The *threads* also known as light as the process *threads* the same process share their execution context, in contrast to processes that are isolated from each other. In particular, all *threads* the same process share the same Me- moire, the same signals and managers in general, the same list of open descriptors? les.

In the nucleus, there is only one data structure representing all objects executing code, the *tasks* described by the structure struct task_struct
[97 , include / linux / sched.h]. This is a particularly long and dependent structure of compiler options. It only represents a small part here.

```
struct task_struct {
    volatile long state; / * Indicator of the activity of the task * /
      [...]
    unsigned int ptrace; / * Indicates whether a tracing task is in progress * /
      [...]
    int prio, static_prio, normal_prio; / * Information priority and scheduling preferences * /

    unsigned int rt_priority;
    const struct sched_class * sched_class;
      [...]
    cpumask_t cpus_allowed; / * Processors on which the task is allowed to run * /

      [...]
    struct list_head tasks; / * List all the tasks * /
      [...]
    struct mm_struct * mm * active_mm; / * Memory of the task, as the core tasks do not have their own memory, they
       borrow those of the last user to have executed task, hence the distinction between mm and active_mm * /

      [...]
    int EXIT_STATE;                          / * State in which the task is complete, this information is
       available to the parent task after the death of a task * /
    int exit_code, exit_signal;
    int pdeath_signal; / * Signal from one task to its Tasks-? Daughters when she dies, optional * /

      [...]
    pid_t pid; / * Identi? Ant task * /
    pid_t tgid; / * Identi? Ing group the task * /
      [...]
    struct task_struct __rcu * real_parent; / * Task actually having given birth to the current task * /

    struct task_struct * __rcu relative; / * Task to which the current task should report his death * /

    struct list_head children; / * List of Tasks-? Girls * /
    struct list_head sibling; / * Entering the Tasks- list? Daughters of the parent task * /

    struct task_struct * group_leader; / * Task of the same group as the task
```

```
        current which identi? ant is identi? ant group, it is designated as group leader (this may be the current task itself) *
        /
    struct list_head ptraced; / * List of tasks followed with ptrace by the current task * /

    struct list_head ptrace_entry; / * Entry in the list of tasks followed by the parent task, if any * /

     [...]
    struct list_head thread_group; / * List of the same group tasks * /
    struct list_head thread_node; / * Entry in the list of the same group tasks * /

     [...]
    cputime_t utime, stime, utimescaled, stimescaled; / * Various counters to measure the execution time of the
        task * /
    cputime_t gtime;
     [...]
    unsigned long nvcsw, nivcsw; / * Counters of the number of context switches for the task * /

    u64 start_time; / * Start time of the task * /
     [...]
    / * Process credentials * /
    const struct cred __rcu * real_cred; / * Authorities actual task (see subsection 3.2.2 ) * /

    const struct cred __rcu * cred; / * Authorizations? Chees task * /
    tank comm [TASK_COMM_LEN]; / * command line that started the task * /
# ifdef CONFIG_SYSVIPC
    struct sysv_sem sysvsem; / * Information on System V semaphores owned * /

    struct sysv_shm sysvshm; / * Information shared memories possessed System V * /

# endif
# ifdef CONFIG_DETECT_HUNG_TASK
    unsigned long last_switch_count; / * Detector blocked tasks, it is a counter of the number of times the task was
        selected by the scheduler to run * /

# endif
    struct fs_struct * fs; / * Information on the system? Les, including the current working directory * /

    struct files_struct * files; / * List of descriptors? Open files * /
    struct NSProxy * NSProxy; / * Namespaces whose task is part * /
    struct signal_struct * signal; / * State signals * /
    struct sighand_struct * sighand; / * Managers installed signals * /
    sigset_t blocked, real_blocked; / * Sets of blocked signals * /
    sigset_t saved_sigmask; / * Mask of saved signals * /
    struct sigpending pending; / * Signals received and awaiting processing * /
     [...]
    struct thread_struct thread; / * State of the current job saved each time the task is stopped by the kernel. This
        condition is necessary to reset the CPU in the specific state where the job was interrupted the next time it is
        selected by the scheduler, this structure is de? Ned so di? Erent depending on the architecture. * /

};
```

The kernel scheduler knows only the tasks that correspond to rather *threads* only process. To add to the confusion, the kernel code sometimes uses

Table 3.2 - Resources can be shared between a job and the new job it creates shareable element

	Description
Address Space	Sharing all memory areas except the battery which is always treated separately and local to each task
System? Le	Sharing of information about a system - files, for example, the root and the current directory
Descriptors? Le	Sharing descriptors? Le open
signal handler list sharing installed signal handlers Ptraçage	If the parent task is ptracée, the new task is also
relative	Sharing the parent task; in other words, the task creating the new task does not become the parent of the latter but his sister
thread	Sharing the same group of tasks, the new job created corresponds to a new *thread* the same process as the current task
System V semaphores	Sharing System V semaphores (used for the syn- chronization between tasks)
O Context Sharing the same information for ma-	nipuler devices via their records or other direct access methods
namespace "System? le"	Sharing the same system? Virtual files (same root, the same tree? Files visible)
Namespace "CGroup"	Sharing the same limitations and quotas on re- sources (such as processors, memory, etc.)
namespace " UTS "	Sharing information on the name of the international machine and other identi? Cation network
namespace " CPI "	Sharing information on CPI System 5; a CPI is accessible via the key that within the namespace where it was created
Namespace "Users"	Sharing users registered in the system
namespace " PID "	Sharing the list of running tasks
Namespace "Network"	Sharing of network interfaces, the routing table, etc.

The namespaces are shared by default. The other elements are not shared if the system call fork is used. Only the system call clone newer, allows finely control the shared items.

word " *thread* " (as in threads_struct) sometimes the word 'process' (as in pid which is short for *Process Identi? Er)* to designate actually a task. This is due to the fact that originally the nucleus did not know the processes and no support was provided in the kernel for *threads* . These were implemented in user space. The introduction of scheduling *threads* in version

2.0 kernel allowed a gain in performance at the cost of increased code complexity, and confusion among novice programmers system trying to understand the core.

Tasks can be grouped: tasks sharing the same field tgid (*thread group identi? er)* are part of the same group. The identi? Ant tgid is equal to the identi? ing of the task group leader, usually the first task of this group. Conceptually, a task group corresponds roughly to a process. In fact, Linux generalizes concepts and processes *thread* enabling two tasks to share some of their resources (presented in table 3.2) While maintaining other strictly separated. A task can for example share with another the table descriptors? Les, or her memory or semaphores, without sharing the rest. Thereafter, the system call unshare allows

de-share this common condition, duplicating the structures concerned. This model is richer than the dichotomy *thread* / process but it is rare that the user programs in pro? tent full. However, some major libraries use this mechanism. There are a few restrictions on shareable resources. In e? And the UNIX model and standards that have followed dictate that when a signal is sent to a process, all *threads* can receive it. Therefore, the kernel ensures that when two tasks share thesame identi? Ant group (that is to say, they represent two *threads* the same process), they share their signal handler, which also involves sharing their memories.

permissions

Permissions are information associated with an object and including [97 ,? the Documents / security / credentials.txt]:

- information about its owner and the owner group;
- access rights associated with it;
- the permissions granted to it to act on other objects in the system;
- a security condition required for a module LSM , If applicable.

The term "authorization" is an imperfect translation from English *credentials* evoking quali cations or diplomas of a person who would appear for a job; in the same way that a process must have su permissions? cient kernel to open a? le. The structure struct cred, associated with the processes, is de? ned in include / linux / cred.h . All *threads* a process share the same structure permissions. The other system objects such as? Files have a smaller structure permits because they do not have, for example, *capabilities.*

In contrast,? Executable files have a unique feature that is the couple of bits SUID / SGID which makes the process running this? le permissions owner / group owner? le.

```
struct {cred
      [...]
    kuid_t uid; / * User actual owner * /
    kgid_t gid; / * Group actual owner * /
    kuid_t suid; / * User saved owner * /
    kgid_t sgid; / * Group owner saved * /
    kuid_t euid; / * owner user? ket * /
    kgid_t egid; / * Group owner? ket * /
    kuid_t fsuid; / * User used for operations on the? Le * /
    kgid_t fsgid; / * Group used for operations on the? Le * /
      [...]
    cap_inheritable kernel_cap_t; / * Capabilities process- them? Ls inherit * /
    kernel_cap_t cap_permitted; / * Capabilities that can be possessed * /
    kernel_cap_t cap_effective; / * Capabilities that can be used * /
    kernel_cap_t CAP_BSET; / * Capacity that is allowed to pass through inheritance * /

      [...]
# ifdef CONFIG_SECURITY
    void * Security; / * Safety construction LSM * /
# endif
    struct user_struct * user; / * Real user responsible (for quota management in particular) * /

    struct user_namespace * user_ns; / * Namespace in which this structure is used * /

    struct group_info * group_info; / * Additional groups critical permissions on? le * /

      [...]
};
```

3.2.3 memory data structures

The memory of each process is an information container. In e? And is within the memory that is allocated to a process may retrieve the contents of a? Le when reading or preparing a message to be sent via a? The message for example. In addition, the memory buffers all input-output operations to files? And hardware devices. In this section, we describe how the memory is divided into pages, organized and assigned to process.

Page cache

When a process reads or writes in a? File stored on a disk, this does not automatically translated into an input-output to the physical device. In e? And a request to said device takes a relatively long time compared to a memory access, due to the material itself. Read from a magnetic hard disk requires particular writing in a register of the microcontroller of the hard drive, which must then interpret the request, move the playhead disks, e? Ectuer playback to retrieve data in a small in memory and then? n copy the data in the main memory for the processor can operate. So there is usually a mechanism

Cache: A portion of the memory is dedicated to the representation of the content of files?. When? Le is read for the first time, a normal request is e? Ectuée to the disk and part of the contents of the? Le is repatriated in the cache. The readings and successive writing are made directly from the cache, which acts as a *proxy* the disc. If a reading is about a part of? Le absent from the cache, a new request to the disk is made to populate. Naturally, the cache has a limited size and sometimes it is necessary to displace some? Les. The core carefully chooses the? Le to eliminate according to the last time they were used and the likelihood of that again in the near future. Like the rest of the memory, the cache is divided into pages of a certain size (the next section describes the concept of paging in more detail). Each page is handled independently and has a status to trace its use. In particular, a page is declared "dirty" if it was written in the cache but modi? Cations have not yet been passed on the physical medium of the le, the disc. It's a *thread* kernel which asynchronously copying the cache on the disks, making sure not to impact system performance while trying to protect user data. In e? And leaving too many dirty pages exposes users to the risk of losing a lot of data in case of unexpected shutdown of the machine. The use of the cache also has the interest uni? Er much of the code files? Systems. ? The physical media of files systems can be varied: hard disk, network storage, memory (for temporary files systems?), Etc. but almost all can bene? t from the cache, a notable exception being the pseudo-systems? les for which the reading of a? le is in fact the output of a kernel function. So cache management is common and files? systems only have to worry about organizing the storage and to the interface between the driver of the storage device and the system? virtual files. The cache is not however benefit? As for all use cases and it is possible for applications that wish to bypass.

Memory Process

The processes are isolated from each other. They can directly access only their own memory. The kernel applies this insulation by introducing a level of indirection in the manner in which memory addresses used by the process are interpreted: the *paging.* The physical memory of the machine is divided into a set of pages of a certain size, usually 4KiB. If an address is written in 32 bits and reference a particular byte, it is composed of 20 bits identi? Ant page plus 12 bits (2^{12} o = o = 4096 4KiB) giving the offset within this page, the identification byte? ed[5]. There are two kinds of addresses. Physical addresses refer to the real memory of the machine only the core can be used. The virtual addresses are for their converted by the processor core and the physical addresses of di way? Erent according to the process that is running. This translation is fast because it is usually made by a processor hardware, the core program in advance. This is achieved by maintaining a mapping between virtual pages of each process and the physical pages. The bits indicating the offset within the page remain them, identical. This indirection has two advantages. On one hand, several processes can use the same virtual addresses without having to coordinate. In fact, they can even use

5. At least that is conceptually what happens; the implementation can be more or less sordid, according to the considered architecture.

more virtual memory than there is physical memory in the machine: the missing pages are simply taken on the hard disk rather than in main memory. Then the translation of virtual addresses into physical addresses provides an opportunity to detect errors of manipulation or deliberate attacks processes. For example, if a process tries to use an invalid address, it is easy to spot because its translation into physical address corresponds to any authorized page.

The memory of each process, also called *address space* as it is for all addresses that can use the process is described by a structure
struct mm_struct. The most interesting field of this structure is struct mm_struct
 * mmap which is a linked list of memory areas allocated to the process. A memory area corresponds to a range of virtual addresses that a process is allowed to use. For example, the code of the program section corresponds to a memory region of a process. If two processes share memory pages, shared memory section included in the memory of each process as a memory area (and thanks to the paging mechanism, it may not even start at the same virtual address in both processes). Access permissions are assigned area by area. All areas are legible; However, only the code pages are usually executable and many pages are not modi? able.

? The presence of a cache open files also allows the establishment of an additional read-write mechanism in the files: the *memory mapping*. Projecting a? Le is to reserve a memory area in the address space of a process such that the virtual addresses of the area to be translated into the addresses of the pages of this? File in the cache. In other words, the memory access by the projector processes are translated into direct cache access, which is equivalent to read or write? Le, without making the family system calls read or write. There are several kinds of projection. projections

shared are those actually using the cache pages for both read and change? er, so that all other system processes will modi? cations, as if they were made in writing in the? le with a call type system write. The second type of projection are the projections *private.* They are also on the cover pages, but only *until the first write.* When writing, the pages are copied to the memory of the process, which reads and modi? E so the copy, not the original. The modi? Cations that it brings are not visible to other processes, and it does not see the modi? Cations of other processes in its projection. Far from being an obscure feature of the system, the projection? Memory files is pervasive on Linux. In e? And it is in this way that the processes execute their program. An executable is divided into several sections: the code, static data, constants, etc. Each of these sections is projected in the process memory when running a program. Libraries that the program requires are also treated the same way. This allows a library very often used as the standard C library, to be charged only once in physical memory, and that each process can see it in its own address space. Projection? Le also serves to implement shared memory sections between multiple processes. The section 3.2.4

describes this mechanism.

The structure vm_area_struct is de? ned as follows [97 , include / linux / mm_ types.h].

```
struct {vm_area_struct
    unsigned long vm_start; / * Virtual start address of the area * /
    unsigned long vm_end; / * Virtual address? n of the area * /
    struct vm_area_struct vm_next *, * vm_prev; / * linked list of all memory areas of the process * /

      [...]
    struct mm_struct * vm_mm; / * address space that owns the memory area * /

    pgprot_t vm_page_prot; / * Access rights to the memory area * /
      [...]
    struct anon_vma_chain list_head; / * List of anonymous memory areas (that is to say not being e projections? Ectuées
        since? Files) in the same address space * /

    const struct vm_operations_struct * vm_ops; / * Operations on the memory area * /

    unsigned long vm_pgoff; / * Lag? Le of the projection area of memory * /

    struct file * vm_file; / * File through the zone * /
      [...]
};
```

3.2.4 Structures corresponding to communication channels between processes

Unlike? Les which represent a permanent storage medium of information, the channels Inter-Process Communication (IPC) • are used for communication between processes. They are not intended to persist over time, or to store information, but only to the route. We distinguish four CPI

? Di erent:

- tubes;
- ? The messages;
- shared memories;
- the *sockets* network.

We do not consider here, as it is customary to do so, semaphores in the list of CPI because they have a synchronization feature and not routing information. Nevertheless, they form a hidden channel storage perfectly? Reliable and a significant bandwidth.

For historical reasons, some CPI are duplicated in the Linux kernel. Thus, there is a version of? The messages and shared memories inherited from System V, the owner UNIX system developed by the company AT & T, and a more recent, consistent with the description given in the standard Mobile Op- rating System Interface (POSIX) • . A? N to maintain compatibility with older programs, the kernel must continue to support both interfaces side user process. However, core side, the code of the two API of? the messages and two API shared memory, respectively, are united? ed.

tubing

The tubes, better known as the *pipes* in English, are kernel memory buffers sets presenting the point of view of user processes like? le rather special. The tubes work only if (at least) two processes are connected: a writer and a reader. ? Unlike a normal file, the data can only be read strictly sequentially: a byte written before another is played before. The tubes have a maximum capacity. If the reader tries to read while the tube is empty, or if the writer tries to write into the tube while it is full, the operation stops and the process is asleep while waiting for the other process fill or empty (respectively) the tube. It is also possible to use the tube in non-blocking mode, in this case, if an operation should normally block, n'e it? ectue a reading or partial write and returns its result. The tubes can be appointed, they then have one or entries in the system? Les. These tubes are created by the system call mknod. They can also be anonymous, in which case they are created by the system call smoking pipe.

This call returns two descriptors? Le, one to read from the tube and the other for writing. The habitual use of these descriptors is that the process making the call then clone the process-? Ls is created with a copy of descripeurs of? Les and the two processes can run their code and each communicate via the tube. It is also possible for a process to send a descriptor? Le to another process via a socket. In addition to fields common to all i-nodes, the following structure is attached in each i-node representing a tube in field i_pipe (see structure struct inode page 39).

```
struct {pipe_inode_info
    [...]
   wait_queue_head_t wait; / * Queue in the event of full or empty tube * /
   unsigned int buffers; / * Number of individual buffer tube component * /
   unsigned int nrbufs; / * Number of non-empty buffers in the tube * /
   unsigned int curbuf; / * current buffer index * /
   unsigned int readers; / * Number of readers of the tube * /
   unsigned int writers; / * Many writers of the tube * /
   unsigned int files; / * Number of descriptors? Files referencing the inode tube * /

   unsigned int waiting_writers; / * Many writers blocked in the kernel * /

    [...]
   struct pipe_buffer * buf; / * Group buffering component tube * /
   struct user_struct * user; / * The user who created the tube * /
};
```

shared memories

Share a memory portion is to allocate two or more processes an address range which corresponds to the same physical memory pages. In this way, each process using an address of this beach reads and writes not only in samémoire but also that of other processes automatically. The mechanism is the same as that of memory in the projection? Les, except that the? Le does not have to actually exist on the disk, it can be a? Temporary file

- existing only in memory, without having to return to a disc - and anonymous - do not appear in the tree of files?. We first describe in this section the shared memories POSIX implemented in terms of mechanisms already described in this chapter. Although in reality the oldest, shared memories System V no longer o? It today that overlay the shared memories POSIX .

Establish a shared memory is for each process IMPLIED c:

1. open the same file;
2. project this? Le in memory shared manner.

If it is not desirable that the? Le is accessible via the system? Les (it would for example allow a third process to read and write to shared memory by reading and writing in the? Le), it is recommended create the? le without directory entry through the specific system call? that Linux memfd_create. The only alternative, more portable but less convenient, is to create a? Regular shit, delete (as long as it is opened, it will continue to exist even without directory entry) then check? Er nobody else has had time to open before e? ectuer memory projection. In all cases, we must find a way to share the descriptor? Le with other processes to share memory area. This can be done through inheritance (the process-? Ls inherit descriptors? Shit their parent) or by sending the descriptor via a particular method sockets.

The System V Interface has dedicated system calls for setting up shared memory and detachment. The memory area is not identi? Ed by a le, but a key to CPI A kind of identification? Ant that all processes eager to share the memory area should know. The operations to be performed are:

1. call shmget for the identi? ing a shared area demémoire System V (and create it if it does not exist);
2. call shmat to attach this shared memory area in space address.

Detach demémoire shared area is via the system call shmdt and its destruction by shmctl when ordering IPC_RMID.

Message queues

The? The messages are information containers conceptually SEM lar to mailboxes. When a process opens a? The messages, it is as if he received a key to the mailbox. It can at any time check, list the mail deposited inside, remove or retrieve a letter. Unlike tubes that are strictly sequential, messages are all marked by an integer whose semantics is at the discretion of the process working with the same messages. It is possible for a process to read only messages with a certain identification? Ing or having an identi? Ant or below a certain value. Typically, this number is used to implement message categories, or priority.

The operation of? The messages System V looks a bit like the System V shared memory system call msgget provides identi? ing of? the message and the need to create it. Then, the system calls msgsnd and msgrcv allow respectively to send and receive messages via the? it. In? N, the system call msgctl with the operation IPC_RMID destroyed? the messages.

? The System V messages are de ned by a specific structure?: struct msg_queue [97 , include / linux / msg.h].

```
struct {msg_queue
    struct kern_ipc_perm q_perm; / * Permission of? The message * /
    time_t q_stime; / * Timestamp of the last message sent * /
    time_t q_rtime; / * Timestamp of the last message reception * /
    time_t q_ctime; / * Timestamp of the last change? The * /
    unsigned long q_cbytes; / * Occupation of? In bytes * /
    unsigned long q_qnum; / * Number of messages in? The * /
    unsigned long q_qbytes; / * Maximum size of the? in bytes * /
    pid_t q_lspid; / * Identi? Ant the last process in writing in? The * /
    pid_t q_lrpid; / * Identi? Ant the last process to be read in? The * /

    struct list_head q_messages; / * Number of messages in? The * /
    struct list_head q_receivers; / * Number of Receivers * /
    struct list_head q_senders; / * Number of issuers * /
};
```

The? The messages POSIX are implemented very di? erently and are actually a special type of i-node. The? Them however have a somewhat strange interface. ? Most operations on them, such as creating and destroying, rely on the system of virtual files but reading and writing are implé- mented via two dedicated call system: mq_timedreceive and mq_timedsend. Direct playback via read returns some information about the?, but not the mes- sages that it stores. The? The messages are represented by the structure struct

mqueue_inode_info [97 , ipc / mqueue.c].

```
struct {mqueue_inode_info
     [...]
    struct inode vfs_inode; / * I-node of? The message. Having the address of the inode structure it is possible to find the
       address of the? Structure the messages with arithmetic fairly simple pointers. * /

    wait_queue_head_t wait_q; / * Queue processes wishing to be notified of incoming messages * /

    struct rb_root msg_tree; / * All the? Messages on organized in a red-black tree * /

     [...]
    struct mq_attr attr; / * Information on the state of the? Number of messages, etc. * /

     [...]
    struct user_struct * user; / * The user who created the? It, for quotas * /
     [...]
    struct ext_wait_queue e_wait_q [2]; / * Queues to the process until the? The empties or fills respectively * /

    unsigned long QSize; / * Total size in bytes of the space occupied by the messages * /

};
```

Messages are represented by a structure struct msg_msg de? ned in include / linux / msg.h , originally for? System V but reused for the POSIX . The structure struct msg_msg provided as an information container. We can not predict in advance how long a message will go in? The before being received, or whether he will be let alone by whom. In e? And the? The messages are persistent and a message can be received by a process long after the death of its transmitter (which itself could, similarly, the issue before the receiver n ' exist).

```
struct {msg_msg
    struct list_head m_list; / * Entry in the list of messages? The * /
    long m_type; / * Message type, semantics is free * /
    size_t m_ts; / * Size of the message content * /
      [...]
    void * Security; / * Safety construction LSM * /
      / * The message follows the rest of the memory page is allocated the structure. * /
};
```

sockets network

The *sockets* are generic interfaces representing a bidirectional communication point. A socket can be connected to another on the same machine, thereby forming a communication channel somewhat similar to two opposite directions of the tube, or to a port on a remote machine over the network, or at an interface to inside the nucleus. There are many types of sockets because of the diversity of existing network protocols but it is possible to distinguish three broad categories.

- UNIX domain sockets, from BSD. These sockets can only be used for communications on both processes in the same machine. They are actually implemented as kernel memory buffers.

- INET domain sockets. These sockets implement the IP The communication protocol between computers connected to the Internet. We can assimilate these sockets all those that implement an inter-machine protocol such as Bluetooth, for example. These sockets can absolutely be used for communication between two processes on the same machine, but with lower performance than others CPI .

- The special sockets, which are used to provide a point of access in sateur utili- space for functions implemented in the kernel to access hardware for example. The Netlink sockets allow for example to receive no- ti? Cation of core events, particularly those related to the network or implement small routines compiling statistics on kernel objects. ALG sockets allow to bene? T from the implementation of cryptographic primi- tives in the nucleus, which is particularly useful on machines with a cryptographic coprocessor. The sockets are implemented in two main structures di? Erent. The first, struct socket represents the aspect "? le" of the socket. In e? And sockets are objects of? Virtual filesystem that are handled via descriptor files. System calls read, write, mmap be used in particular. A few

system calls as send, recv are provided as standard, although a little redundant Linux. In? No, some system calls as bind, listen, accept
are required to use sockets and have no counterpart in other types of? les. The structure struct socket is de? ned in include / linux / net.h .

```
struct {socket
    socket_state state; / * Status of the socket (connected, awaiting connection, etc.) * /

      [...]
    shorts type; / * Type socket (packet, fashion greeting, raw mode, etc.) * /
      [...]
    struct socket_wq __rcu * wq; / * Queue variable use as socket * /

    struct file * file; / * A pointer to the descriptor? Le (useful for certain operations, in particular the destruction of the
        socket and the release of resources) * /
    struct sock * sk; / * Representation "network side" of the socket * /
    const struct proto_ops * ops; / * Operations supported by the socket * /
};
```

The second data structure is struct sock one instance is part of each structure socket. This structure is for the network side of the structure
struct socket. It is de? Ned in include / net / sock.h and is particularly long. In this structure lies the field sk_security which can be used by the modules LSM for attaching the sockets of the security attributes in addition to those of the i-node. The structure sock does not depend on the network protocol. Each protocol extends this structure with the fields needed by de? Ning a new structure whose sock is a member.

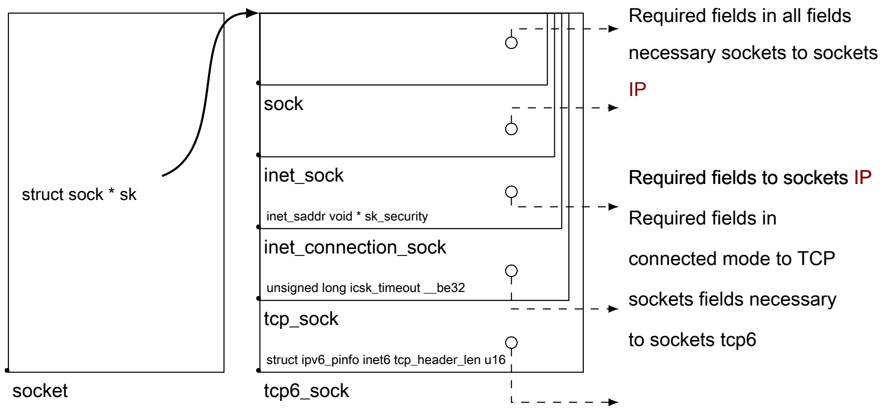

struct sock * sk
socket
sock
inet_sock
inet_saddr void * sk_security
inet_connection_sock
unsigned long icsk_timeout __be32
tcp_sock
struct ipv6_pinfo inet6 tcp_header_len u16
tcp6_sock
Required fields in all fields necessary sockets to sockets IP
Required fields to sockets IP
Required fields in connected mode to TCP sockets fields necessary to sockets tcp6

Figure 3.1 - Nesting and links between the structures of IPv6 TCP sockets.

One can take as an example the structure tcp6_sock presented in? gure 3.1 .
The structure tcp6_sock representing a TCP over IPv6 socket as the first field contains an instance of struct tcp_sock representing a generic TCP socket. The first field of this structure is an instance of struct inet_connection_sock

representing a socket IP running in online mode (as opposed to sockets that can operate without an explicit connection, such as those using the protocol

UDP). This structure in turn contains first field as an instance of struct
inet_sock representing a socket IP generic. This structure contains? Nally as a first field trial sock. This assembly has a consequence very useful if you have a pointer to a structure struct tcp6_sock

, as all nested structures are placed at the beginning of their "contenante" structure, the same pointer can be trivially converted pointer of each other without changing the value. The most specialized structures are therefore extensions of generic structures. It is a kind of ad hoc heritage, in a language that is not object-oriented.

3.3 Conclusion

This chapter exposure does not present a contributionmais rather the fruit of our discoveries in the Linux kernel study. While most books on the Linux kernel - as excellent *Understanding the Linux Kernel [* 6] and *Linux Kernel Development [* 54] - process kernel data structures from the perspective of a beginner programmer system trying to develop its first Linux kernel, we chose the unusual perspective of information flow?. We presented how the abstractions commonly called "process", " *thread* "" Memory "or"? Shit "are actually implemented in the kernel. This presentation serves two purposes. The first, obviously, is to clarify what we mean by the generic term of "information containers". The second, indirect, intended to warn the reader against the temptation to relegate implementation issues tracking greeting secondary. In e? And the code of the kernel, and the Linux instance, has a specific complexity? And that we can not really presuppose what it will be possible to implement in terms security mechanism without knowledge? not the bowels of the system. The next chapter continues this idea by presenting our first contribution: a way to extract the Linux kernel code models for purposes of analysis and visualization?

chapter 4

Linux kernel aided analysis GCC

A? N veri? Er the *framework* LSM is able to serve the correct implementation of greeting information monitors, it is important to rely on formal methods. However, static analysis to bear on the Linux kernel, this poses di? Culty speci? C. We must in particular be able to produce representative and usable model code capturing the position of the hooks provided by the *framework* LSM . In this chapter, we describe the following Kayrebt tools we have developed to this e? And. We also sought in this work to produce generic and reusable tools also for other purposes, such as understanding of the Linux code (which cost us some e? Orts), or even other code bases that the Linux kernel.

4.1 Utilisationdu desmo- faithful compiler to produce the Linux kernel code

4.1.1 Linux kernel code Features

The Linux kernel has several features that make static analysis di? Cult. The most obvious aspect is the size of the source code. The Linux kernel has, in its latest version, about five million lines of code spread over forty-three miles? Les. Of course, no kernel compiled and installed does not use all of that code. The majority of it is in fact the device drivers and disappointed definitions of files? Systems. This contributes to large kernel hardware support, but each machine only needs actually only a part of all this code. The essential features of the kernel (scheduling, virtual filesystem, paging, network stack, etc.) are provided by a small fraction of the total code. However, even this simple fraction alone constitutes a code of impressive size. The second feature is its source code language. In e? And the Linux kernel is coded mostly in C, but not in the ANSI C standard [92]. The extensions of the compiler GCC are explicitly permitted in the core [77]. Until the latest versions, it was also impossible to compile Linux with another compiler GCC . The situation has changed somewhat since the following

LLVM compiler has gained in importance, but the kernel still does not allow compile with another compiler GCC without modi? cations. In addition to the C, part of the kernel code is written in assembly in the clean syntax

GCC . This code can not naturally be regarded in the same way as C, which complicates the writing of static analysis.

Next, the core has the di? Culty to be massively parallelized. In e? And since the disappearance of *Big Kernel Lock* in 2011 [3], There is more critical zone in the core, within which a *thread* could be completely alone to run except on startup. At any time, tens of *threads* di? erent are active. Since the kernel is running mostly in preemptive mode [1] and since the generalization of multi-processor architectures, it faces both false and true parallelism at any point code. This is a di? Culty for static analysis tools because many conditions of competition can occur and the contexts in which they can be triggered are not always easily catchable.

The core also has the characteristic to have multiple entry points. In e? And, while a classical program starts by convention execution from a function called hand in a C program, itself called from

__start, any system call or interrupt handler is for the kernel the beginning of an execution path. This is di? Erent from a server that waits for a connection to start a new process? N to serve the client's request. In e? And, in the case of the nucleus, *thread* user space that makes a system call *becomes,* until the return of the system call, *thread* the kernel space. A special responsibility to remove kernel code before returning to user space all that in the context of implementation of the *thread ,* could reveal information on the ring; except, of course, the expected output of the call and the e? ects edges provided in its speci? cation. The kernel is at the interface between hardware and user process, it also contains the code for processing hardware and software system interruptions. This code is also a kernel entry point. For example, when a process e? Ectue division by zero, an interrupt is immediately lifted by the processor. The running task is suspended and the instruction pointer jumps to the address of a kernel function, scheduled to start. The supported kernel code processing this exception is executed "by surprise" and must respond to process the interrupt (ie, delivering a SIGFPE the offending process) and return the control to the user space. To apply standard assays, it is necessary in practice to consider each system call and interrupt handler as a clean executable, except that the kernel maintains a heavy state shared by all kernel code. Note however that this feature of having multiple entry points is not unique to the core. Android apps are also an entry point for graphics window and per service, the equivalent of several functions except that the kernel maintains a very heavy state shared by all kernel code. Note however that this feature of having multiple entry points is not unique to the core. Android apps are also an entry point for graphics window and per service, the equivalent of several functions except that the kernel maintains a very heavy state shared by all kernel code. Note however that this feature of having multiple entry points is not unique to the core. Android apps are also an entry point for graphics window and per service, the equivalent of several functions hand somehow.

The final complicating factor is the dynamic nature of the nucleus. The kernel development model is now honed for years. Once a kernel version is published immediately began preparing for the next. The first two weeks following the publication of a version are so called uptake period (*merge window* English), because it is during this time that

1. The mode in which a *thread* is likely to see his execution be stopped at any moment? n to allow another *thread* run, as opposed to the cooperative mode where *threads* voluntarily release the CPU.

New features may be subject to the scrutiny of Linus Torvalds, maintainer-in-chief of Linux, in addition to their core. Implementations of these features have generally stayed for some time in the branches of development of deposits of the tests. Then, once the last incorporation period, the version in preparation stabilized and the first release candidate for publication (*release candidate* English) occurred. The next activity is develop- fears intensively and extensively test this version during the coming weeks. Every week, a new release candidate is produced, including the modi? Cations made during the week. Only problem fixes are accepted during this phase and it is strictly forbidden to offer new features. It usually takes six to eight release candidates for the volume of patches decreases su? Ciently and Torvalds judge su kernel? Ciently stabilized to publish a new version, after which the cycle begins again with a new period 'incorporation. It follows therefore that a new kernel version is published every two months to two and a half months, which is a fairly rapid pace.

However, compared to the code of a joint program, the kernel offers certain advantages to facilitate the writing of static analysis. First, only integer variables are used. In e? And arithmetic on real numbers (using the IEEE 754 coding) is not possible in the kernel in general. This is due to the fact that this type of arithmetic requires a set of instructions - and in general a computer unit - dedicated, which is not present on all architectures supported by Linux, especially processors for embedded systems. Even on architectures with the must arbitrate between its use *threads* core and those of the user space, which represents a cost. The types? Oats thus form a category of variables that it is not necessary to consider.

In? N, the kernel does not depend on any third code base. It includes including its own standard library. In e? And it is impossible for an operating system to be able to depend on the performance of loading a library. We therefore have the guarantee when the static analysis to have a complete view of the entire code to be analyzed, without bad can surprise[2].

Utility 4.1.2 compiler to build models

The compiler has the task of transforming the kernel source code, written in C language and dialect in assembler (whose shape depends on the target architecture) into an executable, also called *picture,* the core. The kernel image is a unique compilation of product. It is, in some respects, classical: it is after all a big static archive and its toolchain has not changed for years. In other ways, it is very elaborate. In e? And the kernel supports such dynamic loading of code, in the form of modules. In addition, the code compilation process is extraordinarily configurable. In e? And Linux supports multiple target architectures and many core features can be selected or not to be part of the core. One can thus select systems? Les and device drivers that you want to use,

2. It is possible to load external code to the kernel o? Heaven in the form of modules. It is useful to test new features before proposing them for inclusion in the kernel o? Heaven or to provide support for a device without giving the driver source code. The core is in this case called "dirty" (*tainted* in English). In our analyzes, we have worked on specific cores.

or not to include IPv6 support in the image? nal example. Read the code is a difficult exercise because you have to be able to represent mentally what are the lines of code that are relevant. However, the compiler is still able to distinguish between the selected code for compilation and code left side.

Given the massive size of the code base and di? Culty intrinsic, it is necessary to produce automatic and reproducible models that will support the Linux kernel analysis. The task of the compiler is precisely to give a semantic code. Develop a new syntactic and semantic analyzer of C to extract visualizations is useless, even harmful, because given that C has no formal semantics, there is no guarantee that another analysis than GCC (Which is the compiler "o? Sky" core, in any case, the only explicitly supported by the Linux toolchain) will produce an equivalent result. Our basic idea is that if GCC is capable of producing an optimized executable, while its intermediate representations should also have known? cient information to implement our own static code analysis. This means? E so that GCC must have, in one way or another, internal representations can be used to model both the source code and the executable model. ? Lastly, our last intuition was this: if the internal representations of code GCC can serve? le input to static analysis tools, then these representations should be usable by humans to understand a code base as that of the core. In other words, we are convinced of the potential offered by the compilation of artifacts of the code in terms of models and visualizations of the code.

4.2 Extraction and visualization of graphs? Ow control with Kayrebt

We validated our hypothesis that the compiler is the most appropriate tool to produce models and visualizations of the code that implements the Kayrebt project that has several components:

- *Kayrebt :: Extractor* is a gre? we GCC whose role is to extract, during the compilation of the source code of a C code base graphs? ow control each function.
- *Kayrebt :: Callgraphs* Another gre? is to retrieve pel of belonging graphs of functions of a C code base
- *Kayrebt :: Dumper* is launching a set of scripts *Kayrebt :: Extractor* on the Linux kernel code base, according to di? erent extraction options.
- *Kayrebt :: Globsym* is a set of scripts for extracting the kernel code base all functions de? ned in the core associated with them? le and line de? nition. This knowledge is extracted from the kernel debugging information produced by GCC . The basic functions of data produced is operated by *Kayrebt :: Extractor* a? n to produce enriched graphs of meta data such as the line and the? le of the source code for each node.
- *Kayrebt :: Viewer* is a cross-platform graphical interface for viewing and navigating through the graphs produced by *Kayrebt :: Extractor*
  for a given code base, particularly the nucleus.

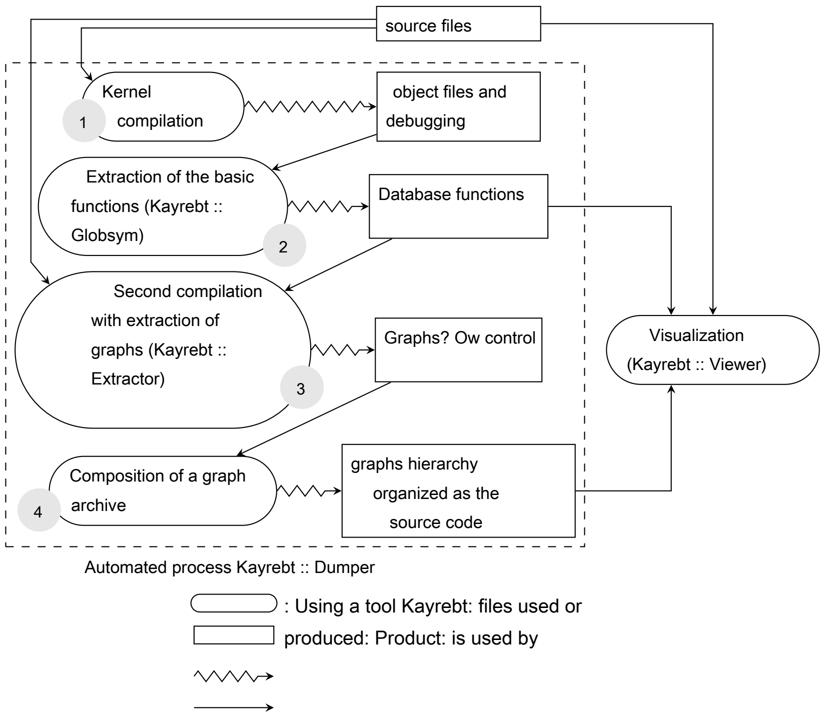

source files
Kernel compilation
1
object files and debugging
Extraction of the basic functions (Kayrebt :: Globsym)
2
Database functions
Second compilation with extraction of graphs (Kayrebt :: Extractor)
3
Graphs? Ow control
Visualization (Kayrebt :: Viewer)
Composition of a graph archive
4
graphs hierarchy organized as the source code
Automated process Kayrebt :: Dumper
: Using a tool Kayrebt: files used or
produced: Product: is used by

Figure 4.1 - Extraction Process graphs ot control of a code base with the following tools Kayrebt

The? Gure 4.1 illustrates the integration of these di? erent tools. Note that Kayrebt :: Call- graphs is a tool to share precedents.

We describe in this chapter Extractor tools and Callgraphs Viewer.

4.2.1 Kayrebt :: Extractor is a gre extraction control ow graphs for GCC?

The suite of compilers *GNU Compilers Collection* supports the injection of gre? ons since version 4.5.0 [88 , 89]. The gre? Ons come in the form of libraries that can be loaded dynamically at runtime. The gre? Ons are passed as parameters to GCC and it is possible to move them any kind of argument. ? The use of gre ons is limited, however: as the compiler is called independently on each *Translation Unit* [3] , it can not easily work on an entire code base and must transformations or manipulations of code are local to the translation unit.

Syntax graphs

Extractor generates a graph by function code. Each node corresponds to an instruction and an arc materializes that a statement may follow another in the code. Each graph contains a single node named "INIT" without

3. In the case of the C language, a translation unit corresponds generally to a? Source file. c.

predecessor, for the entry of the well as a single node "END" function without successor, corresponding to the normal function return. Of course, in the source code, a function normally can be several instructions *return,* but the compiler GCC product instruction? ctive single to reach all these nodes to the compilation. The other nodes without successors represent the deliberate crash of the program (in the case of the nucleus, called the *kernel panics).* It is clear from this that the graphs do not represent the code in the source language. In reality, extracted an intermediate representation of the code built by GCC In a language called Gimple [58]. The code snippet 4.1 shows the function code vfs_llseek

extracted from the kernel. This function implements the system call llseek, for moving the playback position in a? le. We see in the excerpt that initially, a function pointer is declared. Then, depending on whether the file file considered (as a parameter) supports the operation of displacement of the reading position or not, is a function dependent on the? le system on which file lies, a generic function doing nothing is chosen. The selected function is? Nally called. extract 4.2 represents the same function as interpreted by GCC during compilation. GCC is organized as a succession of stages: each stage e modi code for applying a transformation. This transformation can be in optimizing or can calculate a property code, such as register allocation, or pass code of a representation language to another. In? N, the? Gure 4.2 This graph built by Kayrebt :: Extractor for function vfs_llseek.

```
loff_t vfs_llseek ( struct file * file, loff_t offset, int
        whence) {

    loff_t (* Fn) ( struct file *, loff_t , int );

    fn = no_llseek;
    yew (File -> f_mode & FMODE_LSEEK) {
        yew (File -> f_op -> llseek)
            fn = file -> f_op -> llseek; }

    return fn (file, offset, whence); }
```

4.1 Extract - function code vfs_llseek

The Gimple code represents by itself a graph? Ow control, except that it is not unitary instructions that form the nodes but *basic blocks.* A *basic block* is a sequence of instructions which necessarily follow and which are terminated by a conditional jump to several other *basic blocks.* As can be seen in the extract 4.2 , The function contains four *basic blocks* delimited

<Bb N>. Note that the *basic block* 4 is empty and the monitoring unconditionally *basic block* 5. The advantage of this *basic block* is "IHP" node [4] , At the beginning of *basic block* 5. This type of node is used to assign a variable so di? Erent according to the path followed in the function. It is used when the graph? Ow control is called in some form *Static Single Assignment [* 18] (Detailed later in this section). In this instance, fn_1 receives the value no_llseek if the performance comes from

basic blocks 2 or 4 and fn_7 if the performance comes from *basic block* 5. In our graphs such as those of the? Gure 4.2 , The diamond-shaped nodes represent the ends

4. We note these nodes' φ "Later in this thesis.

of *basic blocks.* The arcs represent a guard with conditional jumps, care giving crossing condition. The last node before "END", the rectangle in the graph is the return value of the function.

```
vfs_llseek ( struct file * file, loff_t offset, int whence)
{
    loff_t (* <T26db>) ( struct file *, loff_t , int ) Fn;
    unsigned int _4;
    unsigned int _5;
    const struct file_operations * _6;
    loff_t _11;

    <Bb 2>:
    _4 = file_3 (D) -> f_mode; _4 _5 = & 4;

    yew (_5! = 0)
        goto <bb 3>;
    else
        goto <bb 5>;

    <3 bb>:
    _6 = file_3 (D) -> f_op; fn_7 = _6 -> llseek;

    yew (Fn_7! = 0B)
        goto <bb 5>;
    else
        goto <bb 4>;

    <Bb 4>:

    <Bb 5>:
    #    fn_1 = PHI <no_llseek (2), no_llseek (4) fn_7 (3)> _11 = fn_1 (file_3 (D), offset_8 (D), whence_9
    (D));
    return _11;

}
```

4.2 Extract - intermediate representation of the function vfs_llseek produced by Gimple

Advantages and disadvantages of extracting the graphs from the internal representation of the compiler

The content of the graphs is su? Ciently close to the original code so that we can recognize. This is due to the fact that the C language is relatively low. We did some experiments by applying Kayrebt :: Extractor on code in C ++ and Java. This is possible because GCC treats all languages that accepts as input the same way: they are first converted to the intermediate representation Gimple language, which is that qu'Extractor handles. The results were much less usable because the C ++ and Java are much higher level languages than C. In our test - a simple function that accepts an integer from standard input for a cheap hex on? stdout - we were surprised by the sudden appearance of thirty

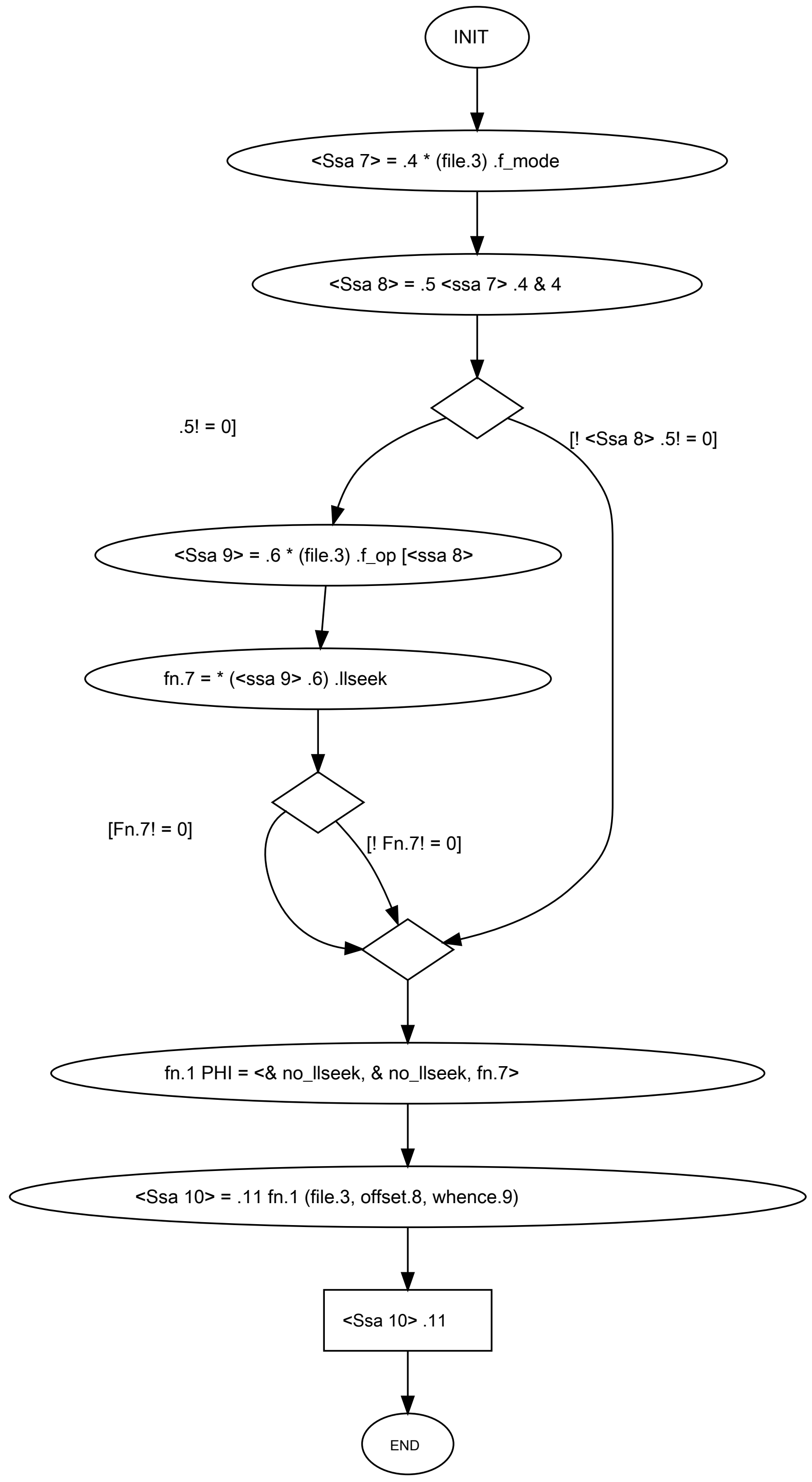

INIT
<Ssa 7> = .4 * (file.3) .f_mode
<Ssa 8> = .5 <ssa 7> .4 & 4
.5! = 0]
[! <Ssa 8> .5! = 0]
<Ssa 9> = .6 * (file.3) .f_op [<ssa 8>
fn.7 = * (<ssa 9> .6) .llseek
[Fn.7! = 0]
[! Fn.7! = 0]
fn.1 PHI = <& no_llseek, & no_llseek, fn.7>
<Ssa 10> = .11 fn.1 (file.3, offset.8, whence.9)
<Ssa 10> .11
END

Figure 4.2 - Graph of the function vfs_llseek

functions to the barbarous names generated by the compiler, and as graphs. The result is di? Cult to see exploitable code. However, it may be interesting for someone wanting to understand the compilation mechanism of a high-level object language.

In the case of C, you get exactly a graph by writing function, or a little less if deemed appropriate compiler to evict some of the functions, and the code is recognizable in the graph. There are, however, di? Erence striking between the original code and the graph. First, it should be noted that Kayrebt :: Extractor is a gre? One of the C compiler, the code that is compiled is not the original one, but the code already transformed by the *preprocessor.* All macros of the original code, for example here in the extract 4.1 macro FMODE_READ, were replaced by their value, in this case 4. This is both an advantage and a disadvantage. In e? And especially in the case of the Linux kernel, developers use and abuse macros to encode iterators on certain data structures, to compile code optionally, to provide di? Erent implementations of the same function as the architecture for which the kernel is compiled, etc. These macros can be di? Cult to follow when reading the source code. However, the graphs contain exactly the code involved in the compilation and therefore that will be part of the executable? Nal. Graphs can therefore be used to show what code is actually used. However, if macros are used as symbolic names to hide arbitrary numerical constants,

In the case of C language as in the excerpt 4.1 , There is also di? E
ences between the original code and the graph. First, in the graph and the code as Gimple, there are many more variables than in the code base. E? And the compiler, among its multiple treatments, simpli? E the code so as to de- cut complex operations involving several operators and dereferencing pointers in unit operations. This involves synthesizing SUP- plementary variables. This e? And is further reinforced by the fact that at the point where the graph is extracted in the toolchain, the code is as *Static Single Assign- ment [* 18], which means GCC applies to version the variable according to the point where they are assigned a value. Thus, in principle, each versioned variable knows only one point of a? Assignment (as if they were constants), which simplifies? E some optimizations in the code. There are exceptions to this principle in particular in the case of global or volatile variables, which explains why GCC applies optimizations much less advanced in using codes. GCC creates nodes *φ* to represent the fact that a certain variable values may have di? erent depending on the path taken to the execution in the function. In the resulting code after compiling these instructions *φ* do not exist. If it is desired that a certain variable *at* is the value of *b* the value of *c* taken according to the way he knew? t in their respective paths, *b* and *c* represents the same memory cell. Thus, where the node is found *φ,* we can give to this common memory box name *at* and the desired result is obtained. The conclusion of this is that use the compiler to extract visual representations of the code can lead to surprises. In e? And must accept all idiosynchrasies the intermediate representation generated by the compiler. It is therefore necessary to choose carefully how the toolchain is desired to insert Kayrebt :: Extractor. In our case, we chose the point of the toolchain where the code is the most optimized while keeping available the graph? Ow control maintained by GCC . In e? And in recent compilation phases, GCC abandons this representation.

Using Kayrebt :: Extractor

GCC is natively capable of producing graphs? ots control graphically, the Graphviz size [27]. Kayrebt :: Extractor also produces graphs in this format. extract 4.3 gives the de? nition of? of the graph figure 4.2 . However, the products have graphs di? Erence. First, our graphs are enriched through the basic functions produced by Kayrebt :: Globsym. In e? And each node corresponding to a static function call contains an argument a hyperlink to the graph of the function called[5]. The basic functions allows us to distinguish the functions called *static* functions *overall*. In C, there can be only one function of a given name in a translation unit. Global functions are common to all units while the static functions are local to their translation unit. In the same code base, so there may be several static functions of the same name in di? Erent? Les. The basic functions allows us in any case to match a function call graph node of the graph of the corresponding function.

```
digraph G {
  graph [file = "fs / read_write.c" line = 252, parameters = "offset,
        whence "]
  29 [shape = "ellipsis" label = "INIT" type = init]; 31 [shape = "ellipsis" label = "<ssa 7> = .4 * (file .3) .f_mode" line

        = 256, filename = "fs / read_write.c" type = assign];
32 [shape = "ellipsis" label = "<ssa 8> = .5 <ssa 7> .4 & 4" line
        = 256, filename = "fs / read_write.c" type = assign]; 33 [shape = "diamond" label = "" line = 256 filename = "fs /
  READ_WRITE
        .  c "type = cond];
  35 [shape = "ellipsis" label = "<ssa 9> = .6 * (file .3) .f_op" line
        = 257, filename = "fs / read_write.c" type = assign]; 36 [shape = "ellipsis" label = "fn.7 = * (<ssa 9> .6)
  .llseek" line
        = 257, filename = "fs / read_write.c" type = assign]; 37 [shape = "diamond" label = "" line = 257 filename = "fs /
  READ_WRITE
        .  c "type = cond];
39 [shape = "diamond" label = ""];
  40 [shape = "ellipsis" label = "fn.1 PHI = <& no_llseek, & no_llseek
        ,     fn.7> "type = phi];
  41 [shape = "ellipsis" label = "<ssa 10> = .11 fn.1 (file.3, offset
        .  8, whence .9) "line = 260, filename =" fs / read_write.c "type = call];

  42 [shape = "rect" label = "<ssa 10> .11" line = 260, filename = "fs /
        read_write.c "type = return];
  43 [shape = "ellipsis" label = "END" type = end_of_activity];
31-> 32 [label = ""];
  32-> 33 [label = ""]; 35-> 36 [label =
  ""]; 36-> 37 [label = ""]; 39-> 40
  [label = ""];

40-> 41 [label = ""];
  41-> 42 [label = ""];
  33-> 39 [label = "! [<Ssa 8> .5 = 0]"];
```

5. In the graph given example, one function call is dynamic, that is to say it is made through a function pointer; in this case it is impossible to know the function called statically Extractor therefore does not place a hyperlink.

```
37-> 39 [label = "[fn.7 = 0]"]; 42-> 43 [label = ""];

29-> 31 [label = ""];
33-> 35 [label = "[<ssa 8> .5 = 0]"]; 37-> 39 [label = "[fn.7 = 0]!"]; }
```

Extract 4.3 - De nition of the function graph? vfs_llseek Graphviz to size

Each node also contains the type statement attribute to which it corresponds, and the? Source file and line of the statement of the source code to which it corresponds. We extract this knowledge of debugging information and diagnos- tic compiler.

In? No, it is possible to assign arbitrary categories to the nodes and arcs of the graphs and their associated formatting commands Graphviz. We used this feature to produce visualizations where nodes corresponding to hooks LSM are highlighted in the graphs with a particular background color. This mechanism allows to apprehend a glance using the features of some API in a code base.

It is necessary to con? Gure Kayrebt :: Extractor to use these features. This is done using a? Le in YAML format which extract 4.4 gives an example.

```
general:
    greedy: 0 url:

        dbfile 'my_db.sqlite'
        dbname 'symbols' categories:

        1 'bgcolor = blue' 2 'textcolor = red'


source_file1.c:
    functions: [ 'function1', 'function2'] match

        (K | m) * alloc. ': 1

source_file2.c:
    functions: [ 'one_more_function'] start_match:

        . * * Spin_lock. ': 2 end_match: [' spin_unlock ']
```

Extract 4.4 - Example of configuration?

We see in this passage all the possibilities designed configuration. The? Le is separated into sections, one for? Le to compile, plus a special section *general* that applies globally. If one wishes to extract all possible graphs of a code base, the designed minimum configuration is to have only the section *general* with the value

greedy 1. Otherwise, it is necessary speci? er for each? le, the list of functions that you want to extract the graph. The "url" gives the name of the? Le and the database produced by Kayrebt :: Globsym. The de? Nition of categories provide the additional attributes of the nodes involved. The sections "source_? Le1.c" and "source_? Le2.c" show two examples of classi? Cation of nodes into categories. In "source_? Le1.c" is assigned to all nodes corresponding to the former regular pressure (k | m) * alloc (. that is to say, the memory allocation functions

Linux kernel) category 1. "source_? Le2.c" is associated with all the nodes located between a node spin_lock and a node spin_unlock (which correspond to synchronization of the functions of *threads* in the kernel) Category 2. This kind of visualized lisation allows this example to see the code that is executed between two locking instructions of a certain data structure, which may correspond to a particularly interesting critical section of code . On the set of graphs of a code base, one can well appreciate a glance if the code is parallelized or if it is constrained by too many locks.

4.2.2 Kayrebt :: Viewer: a display interface free of the graphs produced by Extractor

The collection of graphs produced by Kayrebt :: Extractor is already interesting itself, but viewing the graphs individually is of limited interest in a code base as large of the kernel. It is more interesting to navigate from one graph to another, follow the function calls, etc. For this reason, we have designed other tools Kayrebt result, especially Viewer. Kayrebt :: Viewer is essentially a graphical interface to browse graphs? Ots control. It also depends on? Source files, as well as the basic functions built by Kayrebt :: Globsym. When opening the software, you must give the path of the folder containing the code base, the? Le contains the basic functions and? N the way of graphs folder. This must be organized as the source directory, except that the? Le ".c" source code become folders containing the graphs of functions? Source file in Graphviz format. There is a? Le graph by function, named it.

The? Gure 4.3 provides an overview of the interface. The central panel shows the currently displayed graph. We can handle this view in many ways. We can zoom in and out with the mouse wheel. Graphs can be very variable in size from less than a dozen and a hundred knots. Hover over a node with the mouse did? Che in red, and all nodes reachable from node flew. Double-clicking on a node removes it. It also removes the same time all the nodes and edges of the graph for which there is no longer path from the node "INIT" of the graph. These features help? Lter easily some branches graphs that are not relevant for viewing.

Whenever a chart is opened, the right panel has? Che the source code for the corresponding function. We can thus examine each function and its graph vis-à-vis. When a function has many macros or parts compiled optionally, it can track the? Ow control so? Reliable and helps in understanding the code.

There are several ways to move from one graph to another in the central panel. We first can use the left panel. The upper part has three tabs. The first is a table directly representing the contents of the basic functions. We can? Lter that table by regular expressions of function names or the? Files and source code files. The second tab shows the history graphs session open to use the current Viewer. In? N, the last tab (opened in? Gure 4.3) Is the tree graphs folder. Knowing the organization of sources, we can quickly find the graph of a particular function. Whatever the tab open in the upper part, double-click the name of a function opens its graph.

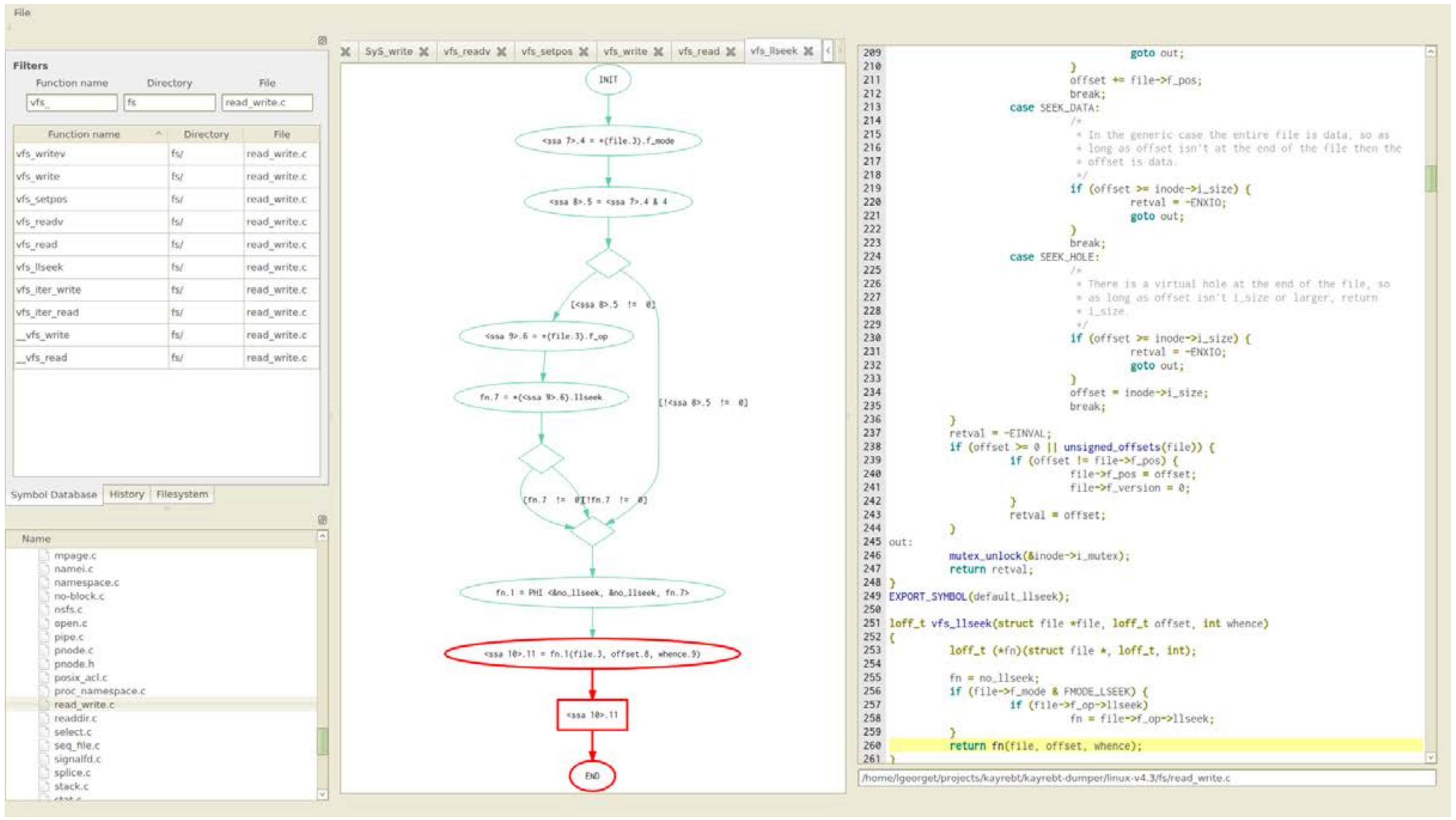


Figure4.3-InterfacedeKayrebt :: Viewer

File
SyS_write
vfs_readv
vfs_setpos
vfs_write
vfs_read
vfs_llseek
INIT
<ssa 7>.4 = *(file.3).f_mode
<ssa 8>.5 = <ssa 7>.4 & 4
[<ssa 8>.5 != 0]
<ssa 9>.6 = *(file.3).f_op
fn.7 = *(<ssa 9>.6).llseek
[!<ssa 8>.5 != 0]
[fn.7 != 0]
[!fn.7 != 0]
fn.1 = PHI <&no_llseek, &no_llseek, fn.7>
<ssa 10>.11 = fn.1(file.3, offset.8, whence.9)
<ssa 10>.11
END
Symbol Database
History
Filesystem

Filters

Function name	Directory	File
vfs_	fs	read_write.c

Function name	Directory	File
vfs_writev	fs/	read_write.c
vfs_write	fs/	read_write.c
vfs_setpos	fs/	read_write.c
vfs_readv	fs/	read_write.c
vfs_read	fs/	read_write.c
vfs_llseek	fs/	read_write.c
vfs_iter_write	fs/	read_write.c
vfs_iter_read	fs/	read_write.c
__vfs_write	fs/	read_write.c
__vfs_read	fs/	read_write.c

```
Name
    mpage.c
    namei.c
    namespace.c
    no-block.c
    nsfs.c
    open.c
    pipe.c
    pnode.c
    pnode.h
    posix_acl.c
    proc_namespace.c
    read_write.c
    readdir.c
    select.c
    seq_file.c
    signalfd.c
    splice.c
    stack.c
```

```
					goto out;
				}
				offset += file->f_pos;
				break;
			case SEEK_DATA:
				/*
				 * In the generic case the entire file is data, so as
				 * long as offset isn't at the end of the file then the
				 * offset is data.
				 */
				if (offset >= inode->i_size) {
					retval = -ENXIO;
					goto out;
				}
				break;
			case SEEK_HOLE:
				/*
				 * There is a virtual hole at the end of the file, so
				 * as long as offset isn't i_size or larger, return
				 * i_size.
				 */
				if (offset >= inode->i_size) {
					retval = -ENXIO;
					goto out;
				}
				offset = inode->i_size;
				break;
		}
		retval = -EINVAL;
		if (offset >= 0 || unsigned_offsets(file)) {
			if (offset != file->f_pos) {
				file->f_pos = offset;
				file->f_version = 0;
			}
			retval = offset;
		}
out:
		mutex_unlock(&inode->i_mutex);
		return retval;
}
EXPORT_SYMBOL(default_llseek);

loff_t vfs_llseek(struct file *file, loff_t offset, int whence)
{
		loff_t (*fn)(struct file *, loff_t, int);

		fn = no_llseek;
		if (file->f_mode & FMODE_LSEEK) {
			if (file->f_op->llseek)
				fn = file->f_op->llseek;
		}
		return fn(file, offset, whence);
}
```

/home/lgeorget/projects/kayrebt/kayrebt-dumper/linux-v4.3/fs/read_write.c

The lower part of the left panel is the source tree. This view has two purposes. First, whenever a graph is selected in a tab in the upper part, the? Le contains the de? Nition of the corresponding function is highlighted in the tree of the lower part. In a particularly large code base as the Linux kernel, it is useful to take over the organization of the sources. Then select a? Le in the tree updates? Lter the first tab in the upper part with the name of the file and its folder. functions can thus quickly through the list of each? le.

4.2.3 Kayrebt :: Callgraphs: a gre it to produce free of call graphs?

Kayrebt :: Callgraphs produces graphs of another type as those produced by Extractor. While the graphs? Ow control enable to evaluate the organ- ization and the paths of execution within a function, the function call graph for understanding how the functions are called together. As Extractor Callgraphs is implemented as a gre? Is to

GCC . In the case of the Linux kernel, it allows us to manage some sensitive cases as functions declared *static* with the same name in several files (we call them "homonyms functions" - recall that several functions can not have the same name in C if at least one of global scope, or if they are to reach *static* and reported in the same translation unit). While a tool could be wrong by building the graph and confuse a function with its namesake, the compiler can not make such a mistake. Nonetheless, penny Callgraphs? Re the same restrictions as all gre? Ons, ie it is run independently on all translation units.

In call graphs, a function is identi? Ed by a name and a file and a line of disappointed nition. These identifying information? Unambiguously ent a function in C because we can not have several functions of the same name de? Ned in the same scope in this language. The gre? Callgraphs it inserts at the beginning of each function toolchain. The generation algorithm of the call graph is very simple: through all the instructions of the function and for each of them corresponding to a static function call (as opposed to a function call through a pointer) it records the fact that being compiled function calls the function of the instruction. Inside the gre? Is it bene? Ts of knowledge GCC and the declaration for the function so we can find. The graph produced is an adjacency list stored in a database. A table lists the functions, while a second contains all couples *<* calling function, called function *>*.

In the case of call graphs, care must be taken to a specific problem: you can call in a translation unit function a de ned in another unit, provided it is not declared *static* but overall. The problem is that when the function call that we see, we can not see the de? Nition, which is in another translation unit, but only its *declaration* in the current translation unit. Therefore, one can not correctly complete the file and line die? Nition. To overcome this problem, we also retain in the database range *static* or global function and apply the following procedure for each meeting of a function call.

- Is recovered identi? Ing of the current function (it was added in the database at the beginning of the function).
- Since the call is recovered the function declaration.
- If the statement shows that the function is *static* in this case, the function is necessarily disappointed denies in the same translation unit.
  - If the function already exists in the database (same name and same shit?) Is recovered identi er in the database?.
  - Otherwise: we add the function in the base and recovered his new ant identical.

- Otherwise, the function is global. ? If it is de ned in another unit of trans- lation:

  - If the function already exists in the database (it only compares the name, because you can not have two global functions of the same name in the same code) is recovered identi er in the database?.
  - If the feature does not already exist in the database: the function is added in the database, without inform his shit or her line of de nition, which are not yet known?. It retrieves the new identi? Ant.
- Otherwise, the function is comprehensive and is de ned in rante LYING translation unit?:

  - If the function already exists in the database: recovering identi er in the database?. It informs the? Le and the declaration line we can now know.

  - If the feature does not already exist in the database: the function is added to the base and recovering the new identi er?.
- the call is added between the current function and that function. Note that you can not directly produce the call graph as and as one travels the code in? CSV file for example. Must necessarily make two passes on the code or use a database management system because we can not know the precise location of the de? Nition of each global function before having analyzed all units translation. To reduce the dependency of our gre? There, we chose the database management system SQLite [41], Which is very light and is based on a single? Le on the disk, the course expense of performance in comparison to heavier systems to install and administer.

Once built the base, it can be analyzed directly or by importing it into a more suitable tool such as the database management system oriented graph Neo4j [60]. For the 4.7 kernel, compiled for the x86 architecture with a con - default configuration, you get a graph of almost seventy thousand nodes and two hundred and thirty thousand bows. The Neo4j database management system is optimized specifically to answer questions like "what are the attainable LSM hooks for this system call? "," What are the common hook these two functions? "Etc.

4.3 Conclusion

The tools Kayrebt :: :: Kayrebt Extractor and Viewer were presented at the conference VISSOFT 2015 [37]. This conference is intended for visualization software in order facilitate understanding, development or its debugging. Kayrebt received very favorably, particularly because it illustrates an innovative approach: the operation of the compiler and build artifacts to the production of software visualizations.

In addition to the good properties that we have already described above (ability to handle a massive code base and complex as the Linux kernel, code interpretation consistent with compiler "o? Sky" to reuse any architecture , designed configuration and version of the code base), the Extractor and Callgraphs tools are also relatively fast although it is not a goal to which we have assigned great importance. We experimented on a machine with a processor 2.10GHz and of 4Gio RAM on which the kernel compilation takes about twenty minutes in normal times. Extraction of graphs? Ow control (a full compile the kernel with gre? Kayrebt :: Extractor is enabled) takes about thirty minutes. Production of the call graph is significantly longer, a little over an hour, but this is mainly due to the use of a SQLite database that we push beyond its nominal use. Use a more efficient database or generate the graph directly into Neo4j certainly reduce compilation time, the price of additional dependencies.

As part of this thesis, tools Kayrebt have helped us to understand both the kernel code and the compiler GCC . In e? And graphs allow you to enter a glance the size of a function and the paths of executions. In particular, it captures a glance paths that reach a certain point of a function. As gre? We Extractor can be placed at various locations in the toolchain, the graphs can show di? Erent intermediate representations of the code built by GCC This allows us to appreciate how

GCC transforms the code during the optimization phase.

chapter 5

Veri? Cation of the offering of LSM hooks for control of greeting information

LSM was introduced into the nucleus in 2001 [104 , 105], With the aim to provide a platform for integrating security modules in the kernel that is totally independent of the functioning and objectives of the modules themselves. In other words, the kernel developers did not want to force the hand of Linux distributions and users in the choice of their security module but still wanted to provide developers a unique and maintainable interface to integrate more easily into the core. This initiative proved ment pay globale- although it has its critics, as the group grsecurity [87] Or RSBAC [68]. These projects are critical including the fact that re-implement their project (before LSM) Cost a lot of e orts; than LSM seems to have been designed for access control in general, and SELinux [85] in particular ; and? n it is almost impossible to have two modules LSM active simultaneously. Some critics are now obsolete because the interface has evolved. Other mo- dules SELinux have been successfully integrated into the sources o? Cial as Smack in February 2008 [97 , E114e473771c848c3cfec05f0123e70f1cdbdc99 commit], To- Moyo in February 2009 [97 , C73bd6d473ceb5d643d3afd7e75b7dc2e6918558 commit] Ap- parmor in July 2010 [97 , Committed cd? 264264254e0fabc8107a33f3bb75a95e981f]. Also, since version 4.2, it is possible to have multiple modules LSM simultaneously, irrespective of the behavior of the modules themselves [97 , E22619a29fcdb513b7bc020e84225bb3b5914259 commit]. If several modules are present, they are all simply consulted in every security decision. It has also made possible the emergence of "small" modules with a unique mission as LoadPin that check? E all? Files loaded by the kernel (such as modules and rmware) come from the same system of? les.

One of these areas of concern remains today: the fact that LSM was primarily designed with the objective of access control and that therefore there is no guarantee that it is appropriate for other purposes, particularly the control? ow of information. However, several security modules as Laminar [76] KBlare [33] EtWeir [67] Are implemented with LSM without any of these approaches discuss the adequacy of this tion *framework* with their objectives; the question is of prime importance.

In this chapter, after a description of the design and implementation of LSM We pose the problem of usability LSM for the implementation of control flow information and we propose our approach to meet them. We conclude that LSM is e? ective appropriate to implement a? ow of information control mechanism, though not without some necessary adaptations due to the di? erence between access control and monitoring greeting.

5.1 LSM and monitors greeting information

LSM provides security module developers two elements:

- additional fields in a number of internal kernel data structures, intentionally unused in the kernel code and left at the disposal of the security modules to store the state necessary to the security decisions;
- points in the kernel code, called hooks, which can be recorded by the security modules functions.

Hooks form a large enough interface. In version 4.7 of the kernel, they number one hundred ninety-eight, listed in Annex AT . At each hook, a security module is free to associate or not function. Whenever a *thread*

meeting a hook during its execution in the kernel, the functions attached to this hook are executed in the order of recording, regardless of the module that recorded them. If one of them returns an error code that bypasses the following stored functions that would normally be called later.

There are two types of hooks. Some are designed to? N to allow security modules maintain their state. It hooks to allocate and deallocate including security fields in structures, or preventing the modules that a certain operation has ended? N to allow updating these security fields. These hooks typically do not have a return value. The other type of hooks is that of authorization hooks. These hooks are placed in the code of system calls, before operations considered sensitive, the security modules may wish selectively ban. These hooks provide a return value in the usual form of a 0 if everything went well and one of the specific error codes? Ed by POSIX otherwise. EPERM, signi? ing "blind permissions? cient." These errors are propagated to the calling function of the nucleus and the userspace code that requested the system call. In this way, the interposer mechanism is perfectly integrated into the kernel. From the perspective of the user process, there is no di erence between a system call failed because of a common mistake - as invalid parameters, a lack of resources or even blind permissions cient detected by? the discretionary access control of the core - and an error reported by one of the security modules.

A feature perhaps surprising the framework LSM is no mechanism for overseas spending decisions by the kernel. In e? And the core has a number of controls in each system call. These controls are generally absolutely necessary, as veri? Er a system call read

'E? ectue course descriptor? shit valid and opened by the calling process. Other controls fall within the traditional discretionary access control Linux, UNIX inherited. The following controls could theoretically be overseas went through a module

LSM no security risk; Nevertheless, the decision was made not to per- put it. The security modules implemented with LSM so can only be applied more restrictions, and never get up.

5.1.1 System Calls causing greeting information

The core is that we analyze the core *vanilla* in its version 4.7. We identi? Ed three hundred and fourteen system calls but not all are responsible for greeting information. Comparing the list of system calls available in the kernel 4.7 on the list drawn up by Hauser in his doctoral thesis [38] On the 3.2 kernel, we can see that no less than seventeen new system calls appeared. The system call interface continuously maintains backward compatibility - there is no way to "break the user space" [95] - but it is far from Gee?. New system calls appear frequently in use by applications or system libraries with specific needs? C.

To list system calls causing greeting information (ta- ble 5.1), We first established a comprehensive list of informa- tion containers to consider:

- the address space of the process;
- ? Regular files;
- tubes;
- ? The System V messages;
- ? The POSIX message;
- UNIX sockets.

several bias atypical in this list can be seen. First, we do not consider the processes themselves as containers as is usual in of greeting control mechanisms but their address space. In e? And this is indeed the memory of the process that contains data, so it is she who is an information container, not the process. In addition, it should be noted that the processes are an abstraction indirectly manipulated by the core because, as we saw in chapter 3 , This ordinance tasks that are closer *threads* that process. They are simply seen as task groups sharing the same signal handler. The distinction is important because, on Linux, it is possible for two *threads* two processes di? erent to share the same memory (however, two *threads* the same process *can not* have a di erent memory); and it is also possible for two *threads* sharing the same memory of the "un-share" (asking the kernel to duplicate). If two *threads* share the same memory, it is natural to consider that they must also share the security label indicating the vows they were the recipients. Therefore, it is easier and safer to attach the labels to memories *threads* rather than

threads themselves.

Second, we do not follow the greeting from or to the outside of the system. Many of greeting information control approaches prevent, in their policy, that external interfaces become tagged, which prevents leakage of information con? Dential to the outside of the system. It would be possible to adopt a similar approach. However, this is outside the scope of our work, which relate only to the *followed* of greeting, not on what it allows monitoring in terms of safety objectives. Hauser has developed Blare [38] a way of

Table 5.1 - RSS caused by the Linux system calls v 4.7 System Call

	Flux
discrete flow	
read	File → memory of the calling process
readv	File → memory of the calling process
preadv	File → memory of the calling process
pread64	File → memory of the calling process
write	Memory of the calling process → take a dump
writev	Memory of the calling process → take a dump
pwritev	Memory of the calling process → take a dump
pwrite64	Memory of the calling process → take a dump
sendfile	File → take a dump
sendfile64	File → take a dump
splice	File → tube
	Tube → take a dump
	Tube → tube
tee	Tube → tube
vmsplice	Memory of the calling process → tube
	Tube → memory of the calling process
recv	socket → memory of the calling process
recvmsg	socket → memory of the calling process
recvmmsg	socket → memory of the calling process
recvfrom	Memory of the calling process → socket
send	Memory of the calling process → socket
sendmsg	Memory of the calling process → socket
sendmmsg	Memory of the calling process → socket
sendto	Memory of the calling process → socket
process_vm_readv	Memory of another process → memory of the calling process
process_vm_writev	Memory of the calling process → memory of another process
migrate_pages	Memory of another process → memory of the calling process
move_pages	Memory of another process → memory of the calling process
fork	Memory of the calling process → memory for a new process
vfork	Memory of the calling process → memory for a new process
clone	Memory of the calling process → memory for a new process

Following Table 5.1. Flow caused by Linux system calls v 4.7

execve.	executable regular file → memory of the current process
execveat.	executable regular file → memory of the current process
msgrcv.	File System V message → memory of the current process
msgsnd.	the current process memory → the System V message
mq_timedreceive. .	File POSIX message → memory of the current process
mq_timedsend.	the current process memory → the POSIX message
continuous flow	
shmat.	POSIX shared memory ↔ memory of the current process
mmap_pgoff.	regular file or device ↔ memory of the current process
. . . .	regular file or device → memory of the current process
mmap.	regular file or device ↔ memory of the current process
. . . .	regular file or device → memory of the current process
ptrace.	the current process memory ↔ memory of another process

follow the colors within a controlled network by transmitting shades with a marking of IP packets. We have not taken this idea because our research is limited to the scale of the operating system, but that would be a possible evolution.

5.1.2 LSM for control of greeting information

All monitors greeting implemented for Linux information that we studied using hooks LSM to implement the monitoring information flow: Laminar [76] KBlare [33 , 40] AndroBlare [2] Flume [51] Weir [67]. One can add to this list Flowx [16 , 17] Which uses the hooks to essentially poly-instantiate information containers according to their security level. It does not really control? Ow of information that our de? Nition but the mandatory access control, although the delimitation becomes a little? Fuzzy at this point. In e? And Flowx separates the objects according to their level of security rather than allowing and prohibiting flow in each case based on the? Ow past. TaintDroid [28] Also uses the hooks to save the labels? Les in the extended attributes of the system? Les, while applying the control? Ow of information at a higher level? N. All these monitors are therefore assume, more or less explicitly, by intercepting all executions via a hook LSM They are able to observe all the vows of information (at least on channels that a? Rm cover). To our knowledge, no work has previously focused on the veri? Cation precisely this assumption, yet crucial. Related work, discussed in chapter 2 section 2.5.4 Have veri? Ed positioning hooks for access to internal kernel data structures, but they leave doubts about our use cases of LSM , Monitoring of greeting. The source code analysis Laminar [75] And KBlare [39] O? Re however some answers. It shows that the addition of hooks LSM

Additional became necessary for e? ectuer track? ow of information in these two tools. Laminar added a total of eight new hooks to allocate and manipulate labels and capabilities, and a hook for allowing or not a process to read the tube in which it tries to write full. These hooks allow the implementation of the model and the closure of the hidden channel formed by the tubes. KBlare added a hook to monitor the detachment shared memories System V to know when the greeting continuous between a process and a shared memory is? N. Weir for its part has not added any hook but covers less CPI UNIX "classic" than the first two. It covers however the Android binder which is a central bus and a major source of greeting in Android systems. In another significant approach Flowx, which does not however implements according to our de? Nition flow control information, the authors have added many hooks. These new hooks allow poly-instantiation of all the information container (process? Les,?

of the messages, etc.) a? n to guarantee a very strong property of non-interference between the process of di? erent levels of security. Although the addition of these hooks is not done with the aim of spreading shade, it should be noted that these hooks are necessary because Flowx seeks to implement policies of greeting more than protecting the data structures of the core. As a result, the hooks added by Flowx, like those offered by Laminar and KBlare indicate that LSM is generally suitable for monitoring flow of information - since, in fact, it is used by many approaches - but partially de cient on some points?.

5.2 Deciding on the correct positioning of brackets

5.2.1 Design of a system call

A Linux system call is encoded in the manner shown in the excerpt 5.1 .
The declaration begins with a macro SYSCALL_DEFINE including the name of the system call and its arguments. This type of statement, unusual, is required to generate at compile a function in the custom kernel to be called from user space, which involves veri? Cation static and conversions of specific types of entry and return function. Some call parameters are marked attribute __ wear indicating that they are pointers from user space, thus referring to the virtual address used by the calling process. This information disappears compilation but can be used by static analysis for veri? Er no marked pointer __ wear is dereferenced by kernel without veri? er its validity. In the context of follow-up? Ow of information, the presence of a pointer __ wear is an interesting heuristic to assume that the system call causes a greeting from the memory of the calling process to other information containers.

A system call is normally composed of three parts:

```
SYSCALL_DEFINE3 (Read, unsigned int , Fd, tank __user * buf,
        size_t , Count)
{
    struct fd = f fdget_pos (fd);
    ssize_t ret = -EBADF;

    yew (F.file) {
        loff_t pos = file_pos_read (f.file); ret = vfs_read (f.file, buf, count, & pos);

        yew (Ret> = 0)
            file_pos_write (f.file, pos); fdput_pos (f); }


    return ret; }
```

Extract 5.1 - Implementation of the system call read

```
int rw_verify_area ( int READ_WRITE, struct file * file, const
        loff_t * Ppos, size_t count) {

    struct inode * inode;
    loff_t pos;
    int retval = -EINVAL;

    = file_inode inode (file);
    yew (Unlikely (( ssize_t ) Count <0))
        return retval;
    pos = * ppos;
    yew (Unlikely (pos <0)) {
        yew (! Unsigned_offsets (file))
            return retval;
        yew (Count> = -pos) / * Both values are in LLONG_MAX 0 .. * /
            return -EOVERFLOW;
    } else if (Unlikely (( loff_t ) (Pos + count) <0)) {
        yew (! Unsigned_offsets (file))
            return retval; }


    yew (Unlikely (inode -> i_flctx mandatory_lock && (inode))) {
        retval = locks_mandatory_area (inode, file, item,
            pos + count - 1, READ_WRITE == READ? F_RDLCK: F_WRLCK);

        yew (Retval <0)
            return retval; }

    return security_file_permission (file,
                READ_WRITE == READ? MAY_READ: MAY_WRITE);
}
```

5.2 Extract - Function rw_verify_area

```
ssize_t __vfs_read ( struct file * file, tank __user * buf,
          size_t count, loff_t * Pos)
{
     yew (File -> f_op -> read)
          return file -> f_op -> read (file, buf, count, pos);
     else if (File -> f_op -> read_iter)
          return new_sync_read (file, buf, count, pos);
     else
          return -EINVAL;
}
```

Extract 5.3 - Function __ vfs_read

- The first part is to analyze the arguments of the system call and veri? Er their validity, then recovering the data structures required for the operation of the system call. It is generally necessary to process synchronizing access to data structures with the other processes executing in parallel, using a kernel locking mechanisms. The veri? Most basic cations are made first, followed by check? Further cations, including the security modules LSM . The system calls can be called many nested functions, as read who calls

  vfs_read who calls herself rw_verify_area and __ vfs_read. Function rw_verify_area (presented in the snippet 5.2) Veri? E that? Le can be read by the process (it is impossible to read? Shit locked or read beyond the? N of? Le) and calls the LSM hook security_file_permission.

  This hook is normally used by the modules LSM to implement veri? cations and prevent additional reading? files under certain conditions. If a check? Cation fails, it is necessary to unlock the structures and undo the operations started in the reverse order of the lock command.

- The second part is the operation of the actual system call. In the case of the system call read, This is the copy of the information? le read to the memory buffer passed by the process. The read operation is performed by the function __ vfs_read Detailed in the extract 5.3 . In the case of system handling calls? Les, perform the requested operation generally equivalent to calling a dependent function of? Le system, via a function pointer in the? Le representative data structure. In the case of read, the kernel calls a function registered in the pointer read

  in the structure f_op Of type struct file_operations the? le.

- In? N, the third part is to undo what is done in the first (as unlock the locked data structures) and to save cer- tain statistics collect traces of access to structures, etc. In? N, the system call must return an integer. This integer is always zero if the call was successful, or a negative return code whose absolute value is speci? Ed by POSIX. For example, EBADF is the value returned by a system call when a descriptor? invalid file it is passed as a parameter. Linux Security Modules can only make a security decision when executing a *thread* reached a hook LSM for it is in these places that only the security module can attach a function. In other words, security modules implemented with *framework* LSM can not observe the full implementation of system calls but only certain items pre disappointed ned. It is therefore possible to detect a greeting and advance the state of the system to record the occurrence of this greeting in a function attached to a hook. Therefore, the position of the hooks is crucial for the proper monitoring of flow?. By studying the system calls generate greeting information and the list of hooks and position, it is possible to know which hooks should be attached a function implementing the monitoring greeting. Equivalently, if there is a system call generating a greeting and information that can be e? Ectuer the greeting without a hook LSM before, then there is a way to dodge the monitoring greeting, so for example the security policies in place. A necessary condition for a follow-up? OK ux is the presence of a hook LSM before each greeting appeals generating system. The presence of a hook after the execution of a greeting is less interesting here because it does not allow for *prevent* an illegal operation in progress. Nevertheless, we will see in the next chapter that post ux hooks are needed for another reason: the management of competition between ux?.

5.2.2 Problems speci? C to the control? Ow of information

Our preliminary research has led us to identi? Er several di? Erence regarding the placement of hooks between the access control and flow control information.

First, the system calls that need to be monitored by the access control are those that create a resource returned to the calling user process, and giving it a way to access an information container. For example, the system call open creates a descriptor? le. This descriptor? File can then be used in the process to ask the system calls read and write. In a sense, the descriptor? Le serves as authorization token allowing the process to access the contents of? Le, the check is made before issuing the token. Sometimes, however, it is necessary to revoke this token. Imagine the following scenario: A user

AT allows another user *B* access to a? le. The process of *B* makes the system call open and sees return a descriptor? le, with whom he starts reading the? le. Thereafter, *AT* his mind and prevents access of? le in *B.* As long as the process *B* does not close the descriptor? le, he can continue to use it to read the? le. For this reason, a hook LSM is also present in read

and write a? n revalidate access? shit during these operations. Revalidation is to veri? Er the conditions met when the system call open are still relevant. The control? Ow of information, instead of the access control requires monitoring system calls causing the greeting as read and write and ignores system calls as open that manipulate the kernel data structures without impacting information containers. We can conclude that only the revalidation hooks are of interest for the control? ow of information.

We made the assumption that all ows require a system call; this- during certain system calls can cause greeting occurring *after* the? n of the system call. We distinguish ux? *discreet* generated entirely during a system call, the greeting *continuous* which are made possible by a system call and ended with another. An example of the first category is ux? Created by the system call read. When the system call ends, the greeting of the? Le to the memory of the process is completed (or does not happen and the system call returned an error). The second category includes greeting inMemory. The system call mmap is used to project a? le in memory. This comprises incorporating into the address space of the process the pages forming the cover of the? Le (that is to say the pages from which the? File is read and written and are synchronized with the physical support for the ? shit from time to time). Once the? Le projected, it is possible for the process to read (and modi? Er, according to the permissions of the memory area) content? Le using only reading-writing in memory, which does not claim to system calls. Another system call, munmap can undo store the screening of a? le. Therefore, we can say that there is a greeting of continuous information between the process and the? Le started by mmap and terminated by munmap. In the case of access control, it is known? Health veri? Er projection and it would be illogical to allow mmap then prohibit munmap, so no hook is necessary in

munmap. In the case of control? Ow, the situation is di? Erent because, as we have seen, the system call munmap allows to know the? n of greeting continuously. Without hook in this call tracking greeting is impossible (except to know the? N of the greeting by other means, which is actually the approach we have adopted in the chapter 6).

5.3 Static analysis check? Ing the right position hooks

The essential property we want veri? Er is the presence of a hook LSM before each instruction causing greeting appeals system concerned. We called this property the *Complete mediation.*

5.3.1 Position of LSM hooks and instructions causing ux

The system call list generating vows, presented in the table 5.1 , has been established Based on previous work Hauser [38] On the implemen- tation of monitor greeting information *KBlare.* The list drawn up by Hauser has been updated to match the 4.7 kernel. It is important to note that non-exhausivité this list does not invalidate our approach because it is not a parameter of our analysis, each system call being independently analyzed.

The position of the hook LSM can be automatically extracted by studying the call graph functions of system calls. We used to do this Kayrebt :: Callgraphs. This gre? We Compiler GCC presented in Chapter 4 calculates and export, during compilation, the graph of calls to each function is to permit to study them with the right tools; in our case, a database oriented graphs, Neo4j. The determination of instructions constituting the? Ow of information itself is much? Fuzzy in general. In e? And unlike hooks LSM Here we encounter several problems:

- E? Ectuer a greeting is not generally an atomic operation. So there is not a single statement but a branch of code that generates the ux?. In this case, it is natural to consider the first instruction of the branch as a starting point ux? And to seek a hook LSM before this point.
- It is not always easy to tell how the greeting is e? Ectué really. When sending a message on a *socket* network, it must be considered that the flow is e? ectué when the data packet has been copied into memory

driver for the network card when the message is sent when it is paid by the recipient? Writing to a local file raises the same type of questions: Is writing made when the cache le is modi ed, when the size of the file is changed, or when the modi cations are synchronized???? on the disk?

- A greeting can take many forms. These modi? Cations memory areas corresponding to information containers such as cache for? Les, memory buffers for network sockets, etc. These modi? Cations can be made via a call to a function such as memcpy,

  or an a? direct assignment, or even the modi? cation of the organization of the user process memory.

A manual analysis enables us to identify? Ux of the first? Generation of points in system calls. We consider that a greeting is e? Ectué when his e? Ects are visible to other processes. For example, when a modified process e memory pages associated with a file in the cache, the other processes are going to read the new data:? So one should consider that the stream is e ectué although synchronization? seek with the disc is not e? ective and will be carried out only later. Other heuristics also guide us. First, the system calls operations generally require a lock or to increment the reference count of manipulated data structures. We therefore know that the important operation of the system call is probably e? Ectuée when the maximum structures are locked. In? N, especially in the case of system calls for? Les or network, operations are delegated to the responsible core sub-module? Le system or network protocol used. We can identify calls to these sub-modules to determine when the flow is done?.

5.3.2 Graphs? Ow control

Our analysis focuses on the code system calls, which we represent, typically in the form of graphs? Ow control. Several points should be noted, that distinguish our approach to most other analyzes. First, our graphs are not C code, the core of program- ming language, but an intermediate representation of the GCC compiler, called Gimple. In fact, our static analysis is implemented as a gre? Is the GCC compiler called Kayrebt :: PathExaminer2 [1] . Graphs are similar to those extracted by Kayrebt :: Extractor (see Chapter 4) But PathExaminer2 works directly from inside GCC On the internal representation of the graph, and not on the extracted graphs Graphviz format. Reasons and benefits of this approach are detailed in subsection 5.7 . The? Gure 5.1 An example graph? ow control as manipulated by our tool. It represents the function vfs_read,

which includes the function rw_verify_area (is recognized in the graph function calls locks_mandatory_area and security_file_permission lines 22 and 27 of the extract 5.2).

In e? And, when compiling a C language code, it is common practice that the compiler optimization called a *inlining* to replace a function call by the body of the latter. Thus, in the graph vfs_read,

the functions rw_verify_area and __ vfs_area are they *inlined,* so that one sees both the passage in the hook LSM and the beginning of the branch where the greeting is

1. There existed a first version since abandoned, hence the name.

generated. For our analysis, we have forced the *inlining* call systems considered, beyond what seemed reasonable to the compiler, so to have the hooks
LSM and generation of points ux? in the same graph. This simplifies? E the analysis but does not change the semantics of the code, because the operation is under the control of the compiler.

We call V all nodes in a graph. We distinguish five types of nodes in the graph, we describe here informally (the de? Ni precise tion is further in section 5.4):

$$V = V_{assign]} V_{same\,]} V_{join]} V_{\varphi]} V_{call}$$

- The nodes of a? Assignment form the subset $V_{assign.}$ In the graph of Figure 5.1 They come in the form < ssa 182> .86 = count.14 'min'
2147478552. The intuitive semantics is that the variable left of the "=" sign is assigned the value of the right expression.
- The nodes of a? Assignment via pointers are the subset $V_{same.}$ They are written eg * < ssa 12> 13 = 0. The variable pointed to by the value to the left of the equal sign, which is a pointer, receives the value of the expression on the right.
- junction nodes are the subset $V_{join.}$ They have no direct counterpart in the C language syntax but simplified? Ent that of the graph, where they are represented by diamonds without text, and facilitate the construction of our model. In e? And these are the only nodes can have multiple incoming or outgoing arcs. These nodes correspond to the beginning and? N *basic blocks,* that is to say sequences of instructions that follow unconditionally. When connecting node has several outgoing arcs, it necessarily corresponds to a conditional branching, and the guards bows are complementary: what are the conditions on the same variables, so that any assessment of these variables is a unique arc whose condition is veri? ed.
- nodes φ, forming the set V_{φ}, are nodes of a? assignment a little par- culiers. They always follow a junction node with multiple incoming arcs represent the fact that at some point program, a variable can have values di? Erent depending on the path taken to reach this point. For example, in the? Gure graph 5.1 The <node ssa 184> .88 = PHI << ssa 183> .87, retval.83, retval.85> is a node φ where the variable left of the sign "= "Receives the value of one of the three variables in the right part, according to the taken branch to reach the junction node just before.
- function call nodes form the subset $V_{call.}$ They represent either a function call, and a? Possible assignment of a return variable, the execution of a assembly code block. In e? And, in our model, these two types of instructions are the same kind of "black box" that can change? Er the memory status of the arbitrary program. Even in

  inlinant some functions, there are still calls in our graphs, including dynamic calls through function pointers. As in Chapter 4 , The graph is as SSA (*Static Single Assignment).*

When a variable of the original source code receives several values successively, it is duplicated by the compiler. When these values di? Erent depend on the path followed, the resulting variable gets its value in a node φ. New variables are also needed to break the complex arithmetic expressions and make the expressions in the right part of the nodes of a? Contain assignment more than one operator (often referred to as "code three addresses" the three addresses are variable back over the two operands). These modi? Cations of the original code are made by the compiler to facilitate its own optimizations and static analysis and we pro? Tones in the de? Nition and implementation of our analysis.

5.3.3 Complete Mediation Property

To check? Er complete mediation of property, we examine each graph individually. Each graph? Ow control exactly is a system call. We consider all *P ATHs* all paths of the graph. Among these paths, we distinguish two particular subsets. The first set

$PATHs_{flows}$ includes all paths beginning at the initial node of the graph (the only node without predecessor, corresponding to the input of the system call), ending to a successor node of the graph without (therefore corresponding to a return from the call) and passing through the node corresponding to an instruction generating a flow?. These paths are those which may correspond to the executions of the function, since its entry to return, responsible for a greeting information. If a call generation system includes multiple points of greeting information independently analyzed all these points. The second subset noted $PATHs_{LSM}$ is that all the roads

$PATHs_{flows}$ comprising nodes corresponding to the hooks LSM located before the node generating the flow?. Naturally, the complete mediation property is trivially satisfied if $PATHs_{flows} = PATHs_{LSM}$. However, this is not the case of all system calls. We notice

$$= P \; PATHs_{flows} \setminus PATHs_{LSM}$$

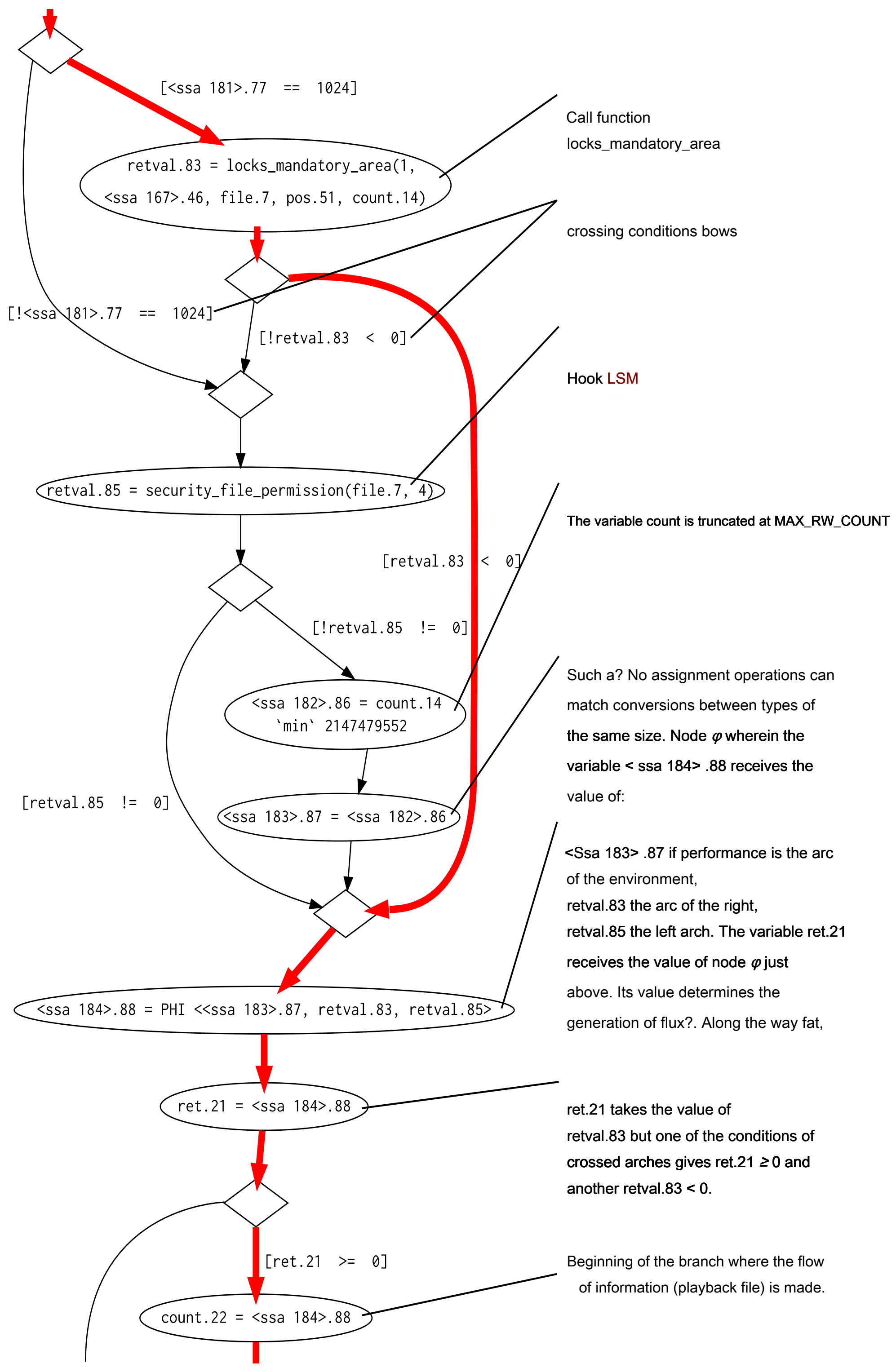
all potentially problematic roads, paths generating greeting unobservable by a monitor greeting constructed with LSM . Some paths P however impossible it is to say that they correspond to any actual implementation of the program; for example because a performance that would borrow this path would pass through both a conditional branch where some boolean variable must be true then another where the same variable is false without being sheave? ected between time (which is true in all cases where the variable obeys the form *Single Static Assignment).* We notice $I \subseteq PATHs$ all impossible paths of the graph. The? Gure 5.1 An example of possible way. The path marked in bold is part of P it avoids the hook LSM and passes into the branch causing ux?. However, it requires that the variable ret.21 has the same value as < ssa 184> .88, itself having the same value as retval.83.

However, this path consists of two arcs, one carrying the strain retval.83 <0
and the other ret.21> = 0, which are incompatible.

We can express the complete mediation property as follows:

Feature 1 (Complete Mediation). Complete mediation is veri? Ed if, and only if, $P \subseteq I$.

The objective of our analysis is to test paths P to prove that they are impossible. However, because of the presence of loops in the code, all P is generally in? no, but it is possible to analyze only acyclic paths (which are number? or in a graph? ni) to conclude on all the roads. We detail in this section 5.4.5 . The? Gure 5.2 summarizes the sets of paths considered and their relationship.


[<ssa 181>.77 == 1024]
retval.83 = locks_mandatory_area(1, <ssa 167>.46, file.7, pos.51, count.14)
[!<ssa 181>.77 == 1024]
[!retval.83 < 0]
retval.85 = security_file_permission(file.7, 4)
[retval.83 < 0]
[!retval.85 != 0]
<ssa 182>.86 = count.14 `min` 2147479552
[retval.85 != 0]
<ssa 183>.87 = <ssa 182>.86
<ssa 184>.88 = PHI <<ssa 183>.87, retval.83, retval.85>
ret.21 = <ssa 184>.88
[ret.21 >= 0]
count.22 = <ssa 184>.88

Call function locks_mandatory_area

crossing conditions bows

Hook LSM

The variable count is truncated at MAX_RW_COUNT

Such a? No assignment operations can match conversions between types of the same size. Node φ wherein the variable < ssa 184> .88 receives the value of:

<Ssa 183> .87 if performance is the arc of the environment, retval.83 the arc of the right, retval.85 the left arch. The variable ret.21 receives the value of node φ just above. Its value determines the generation of flux?. Along the way fat,

ret.21 takes the value of retval.83 but one of the conditions of crossed arches gives ret.21 ≥ 0 and another retval.83 < 0.

Beginning of the branch where the flow of information (playback file) is made.

Figure 5.1 - Example of graph ot control - System Call read

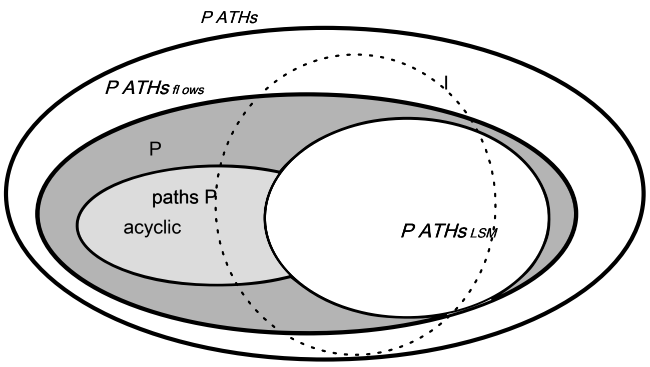


Figure 5.2 - Paths studied in the analysis

5.4 Formalization

In this section, we strive to formally describe the objects of our analysis, that is to say, the graphs, variables, expressions, statements and their semantics. Our approach is structured as follows:

1. The graphs? Ow control form a model code, each graph representing a system call producing greeting information.
2. The nodes of the graph represent the instructions and arches materialize the fact that the execution of an instruction may follow the execution of another.
3. In these graphs, the paths reaching the nodes representing instructions generating greeting information and not passing by nodes representing hooks LSM are problematic because they represent a way to e? ectuer a stream of information that the monitor can not detect. Veri? Er complete mediation property returns to demonstrate that all these paths are impossible.
4. To do this, we assume the existence of a concrete semantics of the code, this semantics is that assigned by GCC the code. We formalize some basic assumptions about the semantics.
5. Next, we construct an abstract semantic us for dinner discrimi- impossible paths of others. We show that this abstract semantics is correct vis-à-vis the concrete semantics, whichever Veri e assumptions formalized the previous point. Thus, our approach is general and could be adapted to other compilers GCC Because it applies whenever the simple assumptions we make are veri? Ed.
6. On the basis of this correction property, then we demonstrate that analyzed and declared impossible path can not be a path taken for a practical implementation of the system call, so it does not violate the property complete mediation.
7. So we unmoyen analyze all paths of length? Niemais this set is in? Or if the presence of loops. We address this problem by the quotient of the assembly according to an equivalence relationship? N to gather in the same equivalence class all the same paths to the number of iterations of their loops close. The normal form of each class is the only acyclic path of this class.

8. We change? Ons abstract semantics so that if the normal form is declared impossible, so we can prove that all the roads of the same equivalence class are also impossible.

9. know? T therefore analyze all acyclic paths to conclude on the impossibility of all paths. This is possible because in a graph? Or there are a number? Or acyclic paths. We have veri? Ed using the Coq proof assistant [93] Demonstrations of our main theorems correction static analysis vis-à-vis the assumptions we have made on the concrete semantics of the code; especially because these demonstrations are heavy and tedious case analyzes. We include in the body of this chapter disappointed definitions and statements in the language of Coq. The script evidence itself is appended B and on the web page of Kayrebt:

https://kayrebt.gforge.inria.fr/proofs.html .

5.4.1 Syntax graphs

After giving an intuitive representation of graphs discussed in the previous section and chapter 4 , We focus here to describe them formally. The graphs? Ots controls are couples (*V E)* with *V* all nodes of the graph and *E* all bows. Each node represents a statement in the Gimple language.

De? Nition 1 (Variables). We notice *V ars* all the variables of the graph that we consider in the analysis. The variables which we do not consider the whole form $\overline{Vars}$. All *V ars* is partitioned in two ways.

- $Vars = Vars_{\mathbb{Z}} \uplus Vars_{ptr}$
- $Vars = Vars_{same} \uplus Vars_{temp}$

The variables in $Vars_{\mathbb{Z}}$ contain whole while variables $Vars_{ptr}$ are of pointer type. The compiler maintains strictly separated (and generates intermediate variables when cast). However, we do not distinguish in our analysis di? Erent types of integers and pointers between them. The variables in $Vars_{same}$ are those living in memory, and can be changed? ed by a? assignment through a pointer, while variables in $Vars_{temp}$ have a value assigned to their creation and the compiler knows they are not modi? ed by e? ects edges. This is the case of the variables that the compiler synthesizes itself, for example. GCC maintains this distinction? n can arbitrarily optimize the variables of the second category. Optimizations on the variables of the first category are more difficult because they can be modi? Ed by e? Ects, or via pointers calculated from known addresses. All variables whose address is taken at a time in the program are $Vars_{same}$, and the variables whose address can be calculated. For example, having a pointer to a field of a structure is equivalent to have a pointer to any other field of the structure.

Note that some variables (forming all $\overline{Vars}$) are not taken into account. These global variables and volatile variables, which can take arbitrary values independently of the program code at any point of execution. We also exclude the structure type variables or union when they use the pointer arithmetic (the compiler can break a structure in as many variables it contains fields in simple cases). Exclude reduced variable the accuracy of our approach because we are losing path information that could help decide they are impossible but maintains its correction, that is to say, we never said not actually possible way. Our analysis intentionally omits many variables as we want to do as little as possible assumptions about the semantics of language operations. In e? And there is no formalization of the semantics attached to the dialect of the C language used to program the Linux kernel by GCC . But this language is notoriously complex, due among other things to the arithmetic of arbitrary pointers, and has many special cases where the semantics of an operation depends in part on the hardware platform. Depend as little as possible of the language semantics source therefore increases the con? Dence in the results of the static analysis and makes it more general. The analysis is built speci? Cally for our needs and results, presented in section 5.6 , Show that it is known? Ciently precise to meet them.

In Coq, we de? Ne two types, one of the variables taken into account and that of unknown variables. ? We de ne three decidable properties on the variables: being distingable, to be or not to be a pointer and addressable or not.

```
Variable Vars: Type .
Variable IgnoredVars: Type .
hypothesis Vars_eq_dec: ∀xy: Vars, {x = y} + {x 6 = y}.

Variable point: Vars → Prop .
Variable addressable Vars → Prop .
hypothesis Addressable_dec: ∀x: Vars, Addressable {x} + { ¬ addressable x}.
hypothesis Pointer_dec: ∀x: Vars point {x} + { ¬ point x}.
```

GCC has an alias noted oracle *O* providing a pointer for the set of variables on which he can possibly point (this information is determined by the compiler essentially according to the variable type).

De? Nition 2 (Oracle alias). The alias oracle is a function:

$$O: Vars_{ptr} \to (Vars_{ptr} \cap Vars_{same})] (Vars_{Z} \cap Vars_{same})$$

Any pointer can be associated with a set of variables on which it is likely to point to. In the worst case, this set is the set $Vars_{same}$ whole. The overall result is partitioned: a pointer can point to either a numeric variable is a pointer, but not both. In addition, the output is necessarily a variable $Vars_{same,}$ that is to say a variable *aliasable.* The output *O* could of course also contain variables *V ars* but we consider the private assembly output *V ars* since we ignore these variables in the analysis.

In Coq, we de? Ne the pointer as a relationship between pointers and addressable va- riables and we assume it is always possible to decide whether a particular pointer can point to a particular variable and whether the pointer points on whole or on pointers.

```
Variable ptoracle: ∀ ( x Vars) (HPTR: point x) (was Vars) (HADDR: Addressable a) Prop .
hypothesis pt_or_not_pt: ∀p HPTR x HADDR, {Ptoracle p HPTR HADDR x} + { ¬ ptoracle p
        HPTR HADDR x}.
```

```
Lemma may_point_to HPTR x p:
  { exists (Hx: addressable x) ptoracle HPTR p} x Hx +
  { ¬ addressable x} +
  { forall (Hx: addressable x) ¬ ptoracle HPTR p} x Hx.

Definition pointers_to_pointers p HPTR: Prop : =
∀ q HADDR, ptoracle p HPTR q HADDR → point q.

Definition pointers_to_non_pointers p HPTR: Prop : =
 ∀ q HADDR, ptoracle p HPTR q HADDR → ¬ point q.

hypothesis cannot_point_to_both:
 ∀ v HPTR, Pointers_to_pointers {v} + {HPTR pointers_to_non_pointers HPTR v}.
```

De? Nition 3 (Expressions and values). There are five types of expressions, each expression of the program with a value decided by the semantics of the language.

- Z is the set of integer constants.
- *V ars* is the set of variables considered in static analysis.
- *{* p | p ∈ V ars ptr}* is the set of pointer dereferencing.
- *{& v | v ∈ V ars same}* is the set of variables of addressing operations.
- *©?* is the set of expressions for which static analysis can not provide the value, that is to say all the expressions that do not fall into the categories above. In fact, all the expressions including a calculation or operation are in this category, as well as variables that are taken into account (*V ars ⊆ ©?).*

In Coq, the set of expressions is an algebraic kind. We are disappointed not ne type of dereferencing operations and taken to address because we take them into account by distinguishing the types of nodes (see below).

```
Variable OtherExprs: Type .

inductive Exprs: Type : =
| int (z: Z)
| var (v: Vars)
| other (o: OtherExprs).
```

The variables are used by the analysis to decide if a path is possible or not. For a way to be possible, we need all the conditional branching component can be taken during the same execution. This is only possible if there is a possible valuation for the variables (for example, should not be in the path of both a node forcing the value of a variable to 0 and a bow asking that the same variable is greater than 1). We de? Ne the notion of

constraints on variables to express the conditions on the value of variables.

De? Nition 4 (Constraints). The constraints form the set *C* they describe predicates on values of the variables. A constraint is a relationship between two variables, a relationship between an integer variable and a constant, or the special constraint true fact denoting the absence of constraint.

$$= C \{\text{true}\}$$
$$\cup Vars \times \{<, >, \leq, \geq, =, = 6\} \times Vars$$
$$\cup Vars_Z \times \{<, >, \leq, \geq, =, = 6\} \times Z$$

The de? Nition Rooster is similar.

```
inductive Ops Type : =
| eq
| neq
| Leq
| Lt
| geq
| Gt.

inductive constraint: Type : =
| constraint1 (v: Vars) (o: Ops) (z: Z)
| constraint2 (v: Vars) (o: Ops) (v ': Vars)
| true.

Lemma Constraint_eq_dec: ∀cc ': Constraint, c = {c '} + { c 6 = c '}.
```

De? Nition 5 (Nodes). A node is an element of all

$$V = V_{assign]} V_{same]} V_{join]} V_{\varphi]} V_{call}$$

The nodes are in the syntax described in the following table, where each list describes a possible syntax. Note that we treat memory addresses as integer constants.

V_{assign}
$x = e$ with:
- $x \in Vars_Z$ and $e \in Z \cup Vars_Z \cup Vars_{ptr}$ * $\cup \{ p \mid p \in Vars_{ptr\}} \cup$ ©?
- $x \in Vars_{ptr}$ and $e \in Z \cup Vars_Z \cup Vars_{ptr}$ * $\cup \{ p \mid p \in Vars_{ptr\}} \cup \{\& y \mid$ there $\in Vars_{same\}} \cup$ ©?
- $w \in \overline{Vars}$ and $e \in Z \cup Vars_Z \cup Vars_{ptr}$ * $\cup \{ p \mid p \in Vars_{ptr\}} \cup \{\& y \mid$ there $\in Vars_{same\}} \cup$ ©? V_{same}
$* p = e$ with:
- $p \in Vars_{ptr}$ and $e \in Z \cup Vars_Z \cup Vars_{ptr}$ * $\cup \{ p \mid p \in Vars_{ptr\}} \cup \{\& y \mid$ there $\in Vars_{same\}} \cup$ ©?
- $p \in \overline{Vars}$ and $e \in Z \cup Vars_Z \cup Vars_{ptr}$ * $\cup \{ p \mid p \in Vars_{ptr\}} \cup \{\& y \mid$ there $\in Vars_{same\}} \cup$ ©? V_{join}

connecting nodes have no specific syntax because they do not correspond strictly to instructions. They serve to simplify? Er syntax graph. Given a junction node v one dice? ne of the arity v

as the number of incoming arcs v.

V_φ

$x =$ IHP $< e_1 \ldots, e_{not} >$ with:

- $x \in Vars_Z \cap Vars_{temp}$ and $\forall i \in \{1 \ldots,$ *born* $_i \in Z \cup Vars_Z \cup$ ©?
- $x \in Vars_{ptr} \cap Vars_{temp}$ and $\forall i \in \{1 \ldots,$ *born* $_i \in Z \cup Vars_{ptr} \cup$ *{& y | there* $\in Vars_{same}\} \cup$ ©?

It may be noted two important features of the nodes φ relative to other nodes truth assignment: the left variable is always a temporary variable and expressions in right part never arithmetic operations. In e? And, GCC in the case of nodes φ always generates the necessary intermediate variables to meet these constraints.

The arity of a node φ is de? ned as the number of expressions in the right part of the node ($e_1 \ldots, e_{not}$). The property is secured by GCC : a knot φ ary *not* in the graph is always preceded or another node φ

ary *not,* or a choice ary node *not.*

V_{call}

Function calls may or may not return a value that can be stored in a variable. A function can have any arguments. It can therefore be written:

$x = f()$ with:

- $x \in Vars$
- $x \in \overline{Vars}$

or

$f()$

or

$x = f(e_1 \ldots, e_{not})$ with:

- $x \in Vars$ and $\forall i \in \{1 \ldots,$ *born* $_i \in Vars \cup Z^* \cup \{p \mid p \in Vars_{ptr}\} \cup$ *{& y | there* $\in Vars_{same}\} \cup$ ©?
- $x \in \overline{Vars}$ and $\forall i \in \{1 \ldots,$ *born* $_i \in Vars \cup Z^* \cup \{p \mid p \in Vars_{ptr}\} \cup$ *{& y | there* $\in Vars_{same}\} \cup$ ©?

or

$f(e_1 \ldots, e_{not})$ with:

- $\forall i \in \{1 \ldots,$ *born* $_i \in Vars \cup Z^* \cup \{p \mid p \in Vars_{ptr}\} \cup$ *{& y | there* $\in Vars_{same}\} \cup$ ©?

We de? Ne similarly Coq all possible nodes using an algebraic kind. Here we restore operations dereferencing and address taken. for example assignPtr_to_Ptr and assignDeref_to_Ptr seem to have the same type but the first case corresponds to a? pointer to pointer assignment (type $p = q$) while the second corresponds to a? assignment of dereference a pointer to another (such as $p = * q$).

```
1 inductive nodes: Type : =
| assignZ_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (E: Z)
| assignVarz_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (e Vars) (Hnptr_e: ¬ point e)
| assignPtr_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (p Vars) (Hptr_p: point p)
| assignOther_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (e: OtherExprs)
| assignDeref_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (e Vars)
    ( Hptr_e: point e) (Heb: pointers_to_non_pointers Hptr_e e)
| assignZ_to_Ptr (p Vars) (Hptr_p: point p) (E: Z)
| assignVarz_to_Ptr (p Vars) (Hptr_p: point p) (x: Vars) (Hnptr_x: ¬ point x)
| assignPtr_to_Ptr (p Vars) (Hptr_p: point p) (q: Vars) (Hptr_q: point q)
    ( Hconsistency: ∀ ( v: Vars) (Haddr_v: addressable v)
        ptoracle q Hptr_q v Haddr_v → ptoracle p Hptr_p Haddr_v v)
| assignDeref_to_Ptr (p Vars) (Hptr_p: point p) (q: Vars) (Hptr_q: point q)
    ( Hq: pointers_to_pointers Hptr_q q)
    ( Hconsistency: ∀ v Haddr_v Hptr_v, ptoracle q Hptr_q v Haddr_v →
        ( forall x Haddr_x, ptoracle v Hptr_v x Haddr_x →
            ptoracle p Hptr_p Haddr_x x))
| assignAddr_to_Ptr (p Vars) (Hptr_p: point p) (a Vars)
    ( Haddr_a: Addressable a) (Hconsistency: ptoracle p Hptr_p has Haddr_a)
| assignOther_to_Ptr (p Vars) (Hptr_p: point d) (e: OtherExprs)
| assign_to_IgnoredVar (w: IgnoredVars) (e: Exprs)
| memZ_to_Ptr (p Vars) (Hptr_p: point p) (E: Z)
    ( hp: pointers_to_non_pointers p Hptr_p)
| memVarz_to_Ptr (p Vars) (Hptr_p: point p) (x: Vars) (Hnptr_x: ¬ point x)
    ( hp: pointers_to_non_pointers p Hptr_p)
| memPtr_to_Ptr (p Vars) (Hptr_p: point p) (q: Vars) (Hptr_q: point q)
    ( hp: pointers_to_pointers p Hptr_p)
    ( Hconsistency: ∀ v Haddr_v Hptr_v, ptoracle p Hptr_p v Haddr_v →
        ( forall x Haddr_x, ptoracle q Hptr_q x Haddr_x →
            ptoracle v Hptr_v Haddr_x x))
| memOther_to_Ptr (p Vars) (Hptr_p: point d) (e: OtherExprs)
| mem_to_IgnoredVar (w: IgnoredVars) (e: Exprs)
| phiZ_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (Hnaddr_x: ¬ addressable x) (E: Z)
| phiVarz_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (Hnaddr_x: ¬ addressable x)
    ( e: Vars) (Hnptr_e: ¬ point e)
| phiOther_to_Varz (x: Vars) (Hnptr_x: ¬ point x) (Hnaddr_x: ¬ addressable x)
    ( e: OtherExprs)
| phiPtr_to_Ptr (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬ addressable p)
    ( e: Vars) (Hptr_e: point e)
    ( Hconsistency: ∀ v Haddr_v, ptoracle e Hptr_e v Haddr_v →
      ptoracle p Hptr_p Haddr_v v)
| phiAddr_to_Ptr (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬ addressable p)
    ( there Vars) (Haddr_y: addressable y)
    ( Hconsistency: ptoracle p Hptr_p there Haddr_y)
| phiOther_to_Ptr (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬ addressable p)
    ( e: OtherExprs)
| callVars (x: Vars)
| callOther (x: IgnoredVars)
| call
| join.
```

De? Nition 6 (Arcs). The arc set is de? Ned as *E = C × V × V.*

An arc is annotated by a constraint in *C* giving crossing condition. The outgoing edges of a node are additional constraints.

In Coq, we de? Ne not the type bows because we handle only one constraint.

5.4.2 Con? Gurations and abstract and concrete executions

It is impossible in the general case to decide if a statically tion exécu- path is possible or impossible because it is a nontrivial program ownership. This is a consequence of Rice's theorem [74 , Corollary B]. However, it is possible to calculate an over-approximation of the possible paths and to exclude certain way much impossible roads. We consider two semantic: the concrete semantics of the code of the system call and abstract semantics which applies to free of graphs. Both are written as semantic relationships transitions between states of the system. In the case of abstract semantics, this state is designed abstract configuration containing constraints on the variables. In the case of the concrete semantics, it is the variable evaluation. We model core concrete performances by *abstract executions* which are the subject of our static analysis. A carry-abstract execution is a sequence of transitions permitted by the abstract semantic and *practical implementation* is its counterpart for the concrete semantics. An abstract execution represents the path of a path of the graph while the corresponding practical implementation is the sequence of instructions in the corresponding core activity. Naturally, we de? Ne the semantic abstract executions based on what we know of the true semantics of system calls, so a path declared impossible by static analysis corresponds to an impossible kernel execution. This proposal 1 page 115 .

Our analysis involves studying each abstract thread has? N to award a designed abstract configuration containing constraints on the variables. To prove that our analysis is correct, the designed abstract configuration of some abstract execution must be consistent with any con? Guration achievable concrete from a corresponding concrete implementation. ? If the con abstract configuration is not possible (for example, it contains incompatible constraints between them) then the path itself is impossible, it can match any concrete implementation.

The con? Gurations are de? Ned formally as follows.

De? Nition 7 (Designed abstract and concrete configurations). The con? Gurations represent the state of the system when running, abstract or concrete.

Designed abstract configuration A designed abstract configuration is part of the whole
$K = \wp(C) \times (Vars_{ptr} \rightarrow Vars_{same} \cup \{>\})$. It is a couple (C, P) taking com-

- C a set of constraints C on the variables;
- P an alias function, giving for each pointer variable to which it points if the latter can be identified? ed univocally, or the special value> indicating that the precise destination of the pointer is not known.

Designed concrete configuration We consider that the program memory is a
together *AT SRFIs* memory boxes, each containing exactly one value. In reality, the elements *AT SRFIs* are a subset of memory addresses, so integers. A designed concrete configuration is a triple ($\gamma, \sigma, \alpha) \in$

Θ or :

- $\gamma: Vars_{temp} \rightarrow \mathbb{Z}$ is a function giving for each temporary variable (variable $Vars_{temp}$) his value ;

- σ*: AT SRFIs* → Z is a function giving for each box memory the variable value stored therein;

- α*: V ars* $_{same}$ → *AT SRFIs* is a function giving for each variable in memory (variables *V ars* $_{same)}$ the memory cell where it is stored. α
is injective, only one variable can occupy a given memory location.

In the designed abstract configuration, function *P* is actually a ra? ment of the alias oracle built by the compiler GCC for its own needs. When a pointer for *p,* we have *P (p) 6 = >* this means? e that our analysis is a more accurate result than this oracle and knows the exact destination pointer *p,* because it takes into account the execution path followed.

In the designed concrete configuration, the first function γ gives the value of off-memory variables. The second gives the program's memory state. It is necessary to distinguish these two functions because in reality, variable *V ars* $_{temp}$ are not modi? ed via memory access. In fact, although part of the graph? Ow control, many of which are optimized by the compiler before producing the executable? Nal. In all cases, they are never modi? Ed by e? And board or via a pointer. The third function? N, gives the memory organization.

In Coq, we de? Ne the con? Gurations similarly, except that the function alias *P* embarks demonstrate its consistency with the alias oracle *O,*
to facilitate demonstrations. In the designed concrete configurations, we divide the functions γ and σ two depending on whether their argument is a pointer or not, is? to facilitate the proof and simplify? er typing. We also use the assumption that the function α never changes (variables do not change addresses once declared, or at least this is transparent from the perspective of the code) into a function in part of the designed concrete configuration.

```
Definition ConstraintSet: Type : = set Constraint.
inductive Target: Type : =
| ptvar (v: Vars) (HADDR: addressable v)
| ptunknown.
Definition PointerMap: Type : = ∀ (x Vars) (HPTR: point x), Target.
Definition PointerMap_is_consistent P: Prop : =
forall (X: Vars) (HPTR: pointer x) (v: Vars) (HADDR: addressable v), P x = HPTR ptvar v
   HADDR → ptoracle x HPTR v HADDR.
Definition AbstractConfiguration: Type : =
  ( ConstraintSet * ( sig PointerMap_is_consistent))% error.

Variable MemoryLocation: Type .
hypothesis MemoryLocation_eq_dec: ∀It ': MemoryLocation, {l = l '} + { l 6 = l '}.
Definition GammaZ: Type : = ∀ (v: Vars) (Haddr_v: ¬ addressable v) Z.
Definition GammaLoc: Type : = ∀ (v: Vars) (Haddr_v: ¬ addressable v) MemoryLocation.
Definition SigmaZ: Type : = ∀ (l: MemoryLocation) Z.
Definition SigmaLoc: Type : = ∀ (l: MemoryLocation) MemoryLocation.
Variable alpha: ∀ (v: Vars) (Haddr_v: addressable v) MemoryLocation.
hypothesis location_is_unique: ∀ Haddr_x Haddr_y x y, x Haddr_x alpha = alpha y
        Haddr_y → x = there.
Definition MemoryState: Type : = (GammaZ * SigmaZ * GammaLoc * SigmaLoc)% error.
Definition gz (m: MemoryState): GammaZ: = fst (fst (m fst)).
Definition sz (m: MemoryState): SigmaZ: = snd (fst (m fst)).
Definition gloc (m: MemoryState): GammaLoc: = snd (m fst).
Definition sloc (m: MemoryState): SigmaLoc: = snd m.
```

We de? Ne also an evaluation function expressions of the code.

De? Nition 8 (Evaluation of a variable). Given a designed concrete configuration $\theta = (\gamma, \sigma, \alpha) \in \Theta$ and a variable $x \in Vars$, one dice? nes $\theta(x)$ as the valuation of x.

We have :

$$\forall x \in Vars\ \theta(x) = \begin{cases} \gamma(x) & x \in Vars_{temp} \\ \sigma(\alpha(x)) & x \in Vars_{same} \end{cases}$$

$$\forall x \in Vars_{ptr}\ \theta(x) \in \bigcup_{z \in Vars_{same}} \{\alpha(z)\} \cup \{0\}$$

with $0 \in AT\ SRFIs$ denoting a special value, that of the null pointer. A pointer whose value is 0 not point to any variables. In C code, a pointer can also point to an invalid address (not corresponding to any variable). We ignore this case (which actually corresponds to a programming error) and consider all the pointers that are not correspond to the value dereferenceable 0.

We de? Ne naturally function assessment in Coq and we formalize the assumption that a pointer can point to some variable that if the oracle alias permits.

```
Definition Valuation (theta: MemoryState) (v: Vars): Z + MemoryLocation.
destruct (Addressable_dec v).
- destruct (Pointer_dec v).
  + exact (Inr (sloc theta (alpha va))).
  + exact (Inl (sz theta (alpha va))).
- destruct (Pointer_dec v).
  + exact (Inr (gloc theta vn)).
  + exact (Inl (gz theta vn)).
Defined .

hypothesis Valuation_is_consistent:
 ∀ ( theta: MemoryState) (p Vars) (Hptr_p: point p)
   ( y: Vars) (Haddr_y: addressable y), p = Rated theta inr (alpha y Haddr_y) → ptoracle p Hptr_p there Haddr_y.
```

De? Nition 9 (Evaluation of an expression). We assume the existence of a func- tion $[\![\cdot]\!]$: $Vars \cup Z\ {*} \cup \{p \mid p \in Vars_{ptr}\} \cup \{\&x \mid x \in Vars_{same}\} \cup ©? \to Z \cup AT\ SRFIs$
associating each term value in the concrete semantics of the language such as:

$$[\![k]\!]_\theta = k \in Z \qquad \text{or } k \text{ is an integer}$$

$$[\![x]\!]_\theta = \theta(x) \qquad \text{or } x \in Vars$$

$$[\![*p]\!]_\theta = \sigma([\![p]\!]_\theta) \text{ or } p \in Vars_{ptr}$$

$$[\![\&x]\!]_\theta = \alpha([\![x]\!]_\theta) \text{ or } x \in Vars_{same}$$

Note that we do not make assumptions about the evaluation of expressions *©? ;*
however, it is assumed that the evaluation ends and causes no writing in the memory.

As we did not die nor to hand operations dereference and address taken, we do not need extra function expressions assessment: the assessment of an integer is the integer itself the evaluation of a variable is given by the presented above function and the valuation of an expression *©?* is unknown.

To express the fact that designed abstract configuration is correct model of a designed concrete configuration, we de? Ne a first compatibility relation between designed concrete configuration and stress.

De? Nition 10 (Compatibility designed configurations). A designed concrete configuration $\theta \in \Theta$
is compatible with a constraint $c \in C$ what one notes θ c if and only if :

c = true

$\vee$ c = (x, S y) $\in V ars_Z \times \{<, >, \leq, \geq, =, = 6\} \times V ars_Z \wedge \theta (x)$ S $\theta (y)$

$\vee$ c = (x, S not) $\in V ars_Z \times \{<, >, \leq, \geq, =, = 6\} \times Z \wedge \theta (x)$ S not

$\vee$ c = (x, S y) $\in V ars_{ptr} \times \{<, >, \leq, \geq, =, = 6\} \times V ars_{ptr} \wedge \theta (x)$ S $\theta (y)$

with S the usual interpretation of the operator taken in {<, > ≤, ≥, =, = 6}.
We extend this relationship to the sets of constraints, slightly abusing ratings.

$$\theta \; C \Leftrightarrow \bigwedge_{c \in C} \theta \; c$$

In Coq, we de? Ne a property named satisfiability to express the compatibility between a designed concrete configuration and a constraint or set of constraints.

```
inductive satisfiability: MemoryState → constraint → Prop : =
| satisfy_true theta:
    satisfiability theta true
| satisfy_constraint_Eq_Z theta x k (Hval: Valuation theta x = inl k):
    satisfiability theta (constraint1 Eq x k)
| satisfy_constraint_Neq_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
    ( Hneq k1 6 = k2):
    satisfiability theta (constraint1 Neq x k2)
| satisfy_constraint_Leq_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
    ( Hneq k1 <= k2)
    satisfiability theta (Leq constraint1 x k2)
| satisfy_constraint_Lt_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
    ( Hneq k1 <k2)
    satisfiability theta (constraint1 Lt x k2)
| satisfy_constraint_Geq_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
    ( Hneq k1> k2 =)
    satisfiability theta (constraint1 Geq x k2)
| satisfy_constraint_Gt_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
    ( Hneq k1> k2)
    satisfiability theta (constraint1 x Gt k2)
```

```
| satisfy_constraint_Eq_Var xy theta
    ( hval: Rated theta x = Rated theta y) satisfiability theta (x Eq
    constraint2 y)
| satisfy_constraint_Neq_Var theta xy k1 k2
    ( Hvalx: Valuation theta x = inl k1) (Hvaly: Rated theta inl y = k2)
    ( Hneq k1 6 = k2):
    satisfiability theta (constraint2 Neq x y)
| satisfy_constraint_Leq_Var theta xy k1 k2
    ( Hvalx: Valuation theta x = inl k1) (Hvaly: Rated theta inl y = k2)
    ( Hneq k1 <= k2)
    satisfiability theta (Leq constraint2 x y)
| satisfy_constraint_Lt_Var theta xy k1 k2
    ( Hvalx: Valuation theta x = inl k1) (Hvaly: Rated theta inl y = k2)
    ( Hneq k1 <k2)
    satisfiability theta (constraint2 Lt x y)
| satisfy_constraint_Geq_Var theta xy k1 k2
    ( Hvalx: Valuation theta x = inl k1) (Hvaly: Rated theta inl y = k2)
    ( Hneq k1> k2 =)
    satisfiability theta (constraint2 Geq x y)
| satisfy_constraint_Gt_Var theta xy k1 k2
    ( Hvalx: Valuation theta x = inl k1) (Hvaly: Rated theta inl y = k2)
    ( Hneq k1> k2)
    satisfiability theta (constraint2 x Gt y).

Definition satisfiability_set_constraints theta C: =
   ∀ c, set_In c C → satisfiability theta c.
```

5.4.3 abstract and concrete Semantics

We assume that there is a concrete semantics of the code, provided by the compi- freezer, can be written in the form of a transition relation between designed concrete configurations.

$$\rightarrow \subseteq \Theta \times V \times C \times \Theta.$$

Intuitively, if (θ_1 *v, c,* θ_2) $\in \rightarrow$, what one notes $\theta_1 \xrightarrow{v,\, c} \theta_2$ this means? e in the actual performance considered, going from the designed configuration θ_1 the designed configuration θ_2 through node *v* his successor by the crossing of the arc labeled by *c,* as shown in? gure 5.3 .

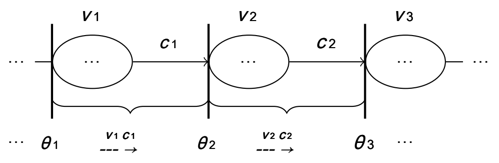


Figure 5.3 - Operation of the concrete semantics: transition includes a knot over an arc

We do not give a de? Nition complete this semantic but we only do some simple assumptions and towns on this semantic, listed below.

We express the properties of the relationship $\rightarrow$ distinguishing each node type.

- $v \in V_{assign}$ of shape $x = e$, with $x \in V\,ars_{temp}$

$$\theta = (\gamma, \sigma, \alpha)_{v,c} \dashrightarrow (\gamma[x \leftarrow J\ e\ K_{\theta]}\ \sigma, \alpha)$$

A a? Single assignment assigns to the variable x the value of the expression e.
With $x \in V\,ars_{temp}$, it is the function γ the designed concrete configuration is modi? ed.

- $v \in V_{assign}$ of shape $x = e$, with $x \in V\,ars_{same}$

$$\theta = (\gamma, \sigma, \alpha)_{v,c} \dashrightarrow (\gamma, \sigma[\alpha(x) \leftarrow J\ e\ K_{\theta]}\ \alpha)$$

With $x \in V\,ars_{same}$, it is the function σ which is modi? ed. $\alpha(x)$ here gives the address of x.

- $v \in V_{same}$ of shape $*p = e$

$$\theta = (\gamma, \sigma, \alpha)_{v,c} \dashrightarrow (\gamma, \sigma[J\ p\ K_{\theta} \leftarrow J\ e\ K_{\theta]}\ \alpha)$$

A a? Ection through a pointer put the value of the expression e in va- riable pointed by p, which is necessarily a variable in memory
$V\,ars_{same}$. So the function σ which is modi? ed.

- $v \in V_{\varphi}$ of shape $x =$ IHP $< e_1 \ldots, e_{not} >$

$$\theta = (\gamma, \sigma, \alpha)_{v,c} \dashrightarrow (\gamma[x \leftarrow J\ e_{not_{path}}\ K_{\theta]}\ \sigma, \alpha)$$

or not_{path} is the index of the arc made to reach the last junction node (this index is maintained along the route of the road, by numbering the bows of each connecting node). One has? Assignment in a knot φ sets the value of an expression at the right side, selected according to the path followed, in x which is always worth $V\,ars_{temp}$ (This restriction is enforced by the compiler). It is always possible to know not_{path} in our case, since we analyze the ways one by one.

- $v \in V_{call}$ of shape $x = f(e_1 \ldots, e_{not})$

$$(\gamma, \sigma, \alpha)_{v,c} \dashrightarrow (\gamma_2\ \sigma_2\ \alpha_2)$$

or $\forall at \in V\,ars_{temp} \setminus \{x\}\ \gamma_2(a) = \gamma(a)$ and σ_2 and α_2 represent a memory state in which no assumptions are made.

A function call (or running an assembly code portion) is likely to change the values of all variables $V\,ars_{same}$

because they could be a? ected, including via a pointer in the func- tion called. However, variables $V\,ars_{temp}$ are protected because the compiler keeps strictly separated in order to enforce its own optimizations. obtained is thus designed concrete configuration a resultant arbitrarily di memory state? erent and a function γ identical to that of departure, except as regards the result variable (always a variable $Vars_{temp)}$ which can contain an arbitrary value to the return of function.

- $v \in V_{call}$ of shape $f(e_1 \ldots, e_{not})$

$$(\gamma, \sigma, \alpha) \xrightarrow{v,c} (\gamma, \sigma_2\ \alpha_2)$$

or σ_2 and α_2 represent a memory state in which no assumptions are made.

The situation is similar in the case where there is no return variable.

- $v \in V_{join}$

$$\theta \xrightarrow{v,c} \theta$$

Nodes choices do not change the designed current configuration because they do not correspond to physical education program has finished, they belong only to the syntax of the graph.

It also has a property arches. An execution can not borrow ter an arc if the crossing condition is not veri? Ed in the designed concrete configuration.

$$\forall \theta_1\ \theta_2 \in \Theta\ \forall v\ \forall \in V c \in C\ \theta_1 \xrightarrow{v,c} \theta_2 \Rightarrow \theta_2\ c \tag{5.1}$$

In Coq, we follow the same reasoning. We assume first the existence of a function transforming a designed concrete configuration in another passage of a knot and then we de? Ne a relationship between designed concrete configurations conditioned by a node and a constraint (supported by the arc following the node). The relationship is veri ed if the final configuration is the result of the updating of the con -???? Initial configuration to the node of the passage and it is compatible with coercion. This models the fact that expression can not borrow a conditional branch if the condition is met. We then list the assumptions that apply to each type of node.

```
Variable ConcreteSemantics ': MemoryState → nodes → MemoryState.
inductive ConcreteSemantics: MemoryState → nodes → constraint →
        MemoryState → Prop : =
  ConcreteSemanticsDef theta vc (Hsatis_c: satisfiability (ConcreteSemantics '
        theta v) c):
  ConcreteSemantics theta vc (ConcreteSemantics ' theta v).

hypothesis csem_assignZ_addr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Haddr_x: addressable x) z ConcreteSemantics ' theta
   (assignZ_to_Varz x Hnptr_x z) = update_sz theta (alpha x Haddr_x) z.


hypothesis csem_assignZ_naddr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Hnaddr_x: ¬addressable x) z ConcreteSemantics ' theta
   (assignZ_to_Varz x Hnptr_x z) = update_gz theta x Hnptr_x Hnaddr_x z.
```

```
hypothesis csem_assignVarz_addr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Haddr_x: addressable x)
   ( e: Vars) (Hnptr_e: ¬point e) ConcreteSemantics ' theta (assignVarz_to_Varz x Hnptr_x Hnptr_e e) =

  update_sz theta (alpha x Haddr_x) (E Hnptr_e ValuationZ theta).

hypothesis csem_assignVarz_naddr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Hnaddr_x: ¬addressable x)
   ( e: Vars) (Hnptr_e: ¬point e)
  ConcreteSemantics ' theta (assignVarz_to_Varz x Hnptr_x Hnptr_e e) = update_gz theta x Hnptr_x
   Hnaddr_x (ValuationZ theta Hnptr_e e).

hypothesis csem_assignPtr_to_Varz:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (e Vars) (Hptr_e: point e)
∃theta2,
  ConcreteSemantics ' theta (assignPtr_to_Varz x Hnptr_x Hptr_e e) = theta2 ∧
  ( forall (Y: Vars), x 6 = there → Rated theta y = Rated theta2 y).

hypothesis csem_assignOther_to_Varz:
∀theta (x: Vars) (Hptr_x: ¬point x) (e: OtherExprs)
 ∃theta2,
  ConcreteSemantics ' theta (assignOther_to_Varz Hptr_x x e) = theta2 ∧
  ( forall (Y: Vars), x 6 = there → Rated theta y = Rated theta2 y).

hypothesis csem_assignDeref_to_Varz_addr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Haddr_x: addressable x)
   ( e: Vars) (Hptr_e: point e)
   ( Hey: pointers_to_non_pointers Hptr_e e) ConcreteSemantics ' theta (assignDeref_to_Varz x Hnptr_x e
  Hptr_e He) =
   update_sz theta (alpha x Haddr_x) ((sz theta) (ValuationLoc theta Hptr_e e)).

hypothesis csem_assignDeref_to_Varz_naddr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Hnaddr_x: ¬addressable x)
   ( e: Vars) (Hptr_e: point e)
   ( Hey: pointers_to_non_pointers Hptr_e e) ConcreteSemantics ' theta (assignDeref_to_Varz x Hnptr_x e Hptr_e He) =
  update_gz theta x Hnptr_x Hnaddr_x ((sz theta) (ValuationLoc theta Hptr_e e)).


hypothesis csem_assignZ_to_Ptr:
∀theta (x: Vars) (Hptr_x: pointer x) (z: Z),
 ∃theta2,
  ConcreteSemantics ' theta (assignZ_to_Ptr x Hptr_x z) = theta2 ∧ ∀( y: Vars), x 6 = there → Rated
   theta y = Rated theta2 there.

hypothesis csem_assignVarz_to_Ptr:
 ∀theta (x: Vars) (Hptr_x: point x) (e Vars) (Hnptr_e: ¬point e)
 ∃theta2,
  ConcreteSemantics ' theta (assignVarz_to_Ptr x Hptr_x Hnptr_e e) = theta2 ∧ ∀( y: Vars), x 6 = there → Rated theta
   y = Rated theta2 there.

hypothesis csem_assignPtr_to_Ptr_addr:
 ∀theta (p Vars) (Hptr_p: point p) (Haddr_p: addressable p)
   ( q: Vars) (Hptr_q: point q) Hconsistency, ConcreteSemantics ' theta (assignPtr_to_Ptr p Hptr_p Hptr_q Hconsistency q)
  =
    update_sloc theta (alpha Haddr_p p) (Q Hptr_q ValuationLoc theta).

hypothesis csem_assignPtr_to_Ptr_naddr:
  ∀theta (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬addressable p)
    ( q: Vars) (Hptr_q: point q) Hconsistency,
  ConcreteSemantics ' theta (assignPtr_to_Ptr p Hptr_p Hptr_q Hconsistency q) = p Hptr_p Hnaddr_p update_gloc theta
    (theta ValuationLoc q Hptr_q).

hypothesis csem_assignDeref_to_Ptr_addr:
  ∀theta (p Vars) (Hptr_p: point p) (Haddr_p: addressable p)
    ( q: Vars) (Hptr_q: point q) Hpointers_q Hconsistency, ConcreteSemantics ' theta (assignDeref_to_Ptr p Hptr_p q
  Hptr_q Hpointers_q
        Hconsistency) =
    update_sloc theta (alpha Haddr_p p) ((sloc theta) (Q Hptr_q ValuationLoc theta)
        ).

hypothesis csem_assignDeref_to_Ptr_naddr:
∀theta (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬addressable p)
    ( q: Vars) (Hptr_q: point q) Hpointers_q Hconsistency, ConcreteSemantics ' theta (assignDeref_to_Ptr p Hptr_p q
  Hptr_q Hpointers_q
        Hconsistency) =
    update_gloc theta p Hptr_p Hnaddr_p ((sloc theta) (ValuationLoc theta q Hptr_q
        )).

hypothesis csem_assignAddr_to_Ptr_addr:
  ∀theta (p Vars) (Hptr_p: point p) (Haddr_p: addressable p)
    ( a: Vars) (Haddr_a: addressable a) Hconsistency, ConcreteSemantics ' theta (assignAddr_to_Ptr p Hptr_p has
  Haddr_a Hconsistency)
        =
    update_sloc theta (alpha Haddr_p p) (alpha has Haddr_a).

hypothesis csem_assignAddr_to_Ptr_naddr:
  ∀theta (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬addressable p)
    ( a: Vars) (Haddr_a: addressable a) Hconsistency, ConcreteSemantics ' theta (assignAddr_to_Ptr p Hptr_p has
  Haddr_a Hconsistency)
        =
    update_gloc theta p Hptr_p Hnaddr_p (alpha has Haddr_a).

hypothesis csem_assignOther_to_Ptr:
  ∀theta (p Vars) (Hptr_p: point d) (e: OtherExprs)
  ∃theta2,
  ConcreteSemantics ' theta (assignOther_to_Ptr p Hptr_p e) = theta2 ∧ ∀y, p 6 = there → Rated theta y = Rated
   theta2 there.

hypothesis csem_assign_to_IgnoredVar:
  ∀theta (w: IgnoredVars) (e: Exprs)
  ConcreteSemantics ' theta (assign_to_IgnoredVar we) = theta.

hypothesis csem_memZ_to_Ptr:
  ∀theta (p Vars) (Hptr_p: pointer p) (E: Z) Hpointers_p, ConcreteSemantics ' theta (memZ_to_Ptr p
  Hptr_p Hpointers_p e) =
    update_sz theta (theta ValuationLoc p Hptr_p) e.

hypothesis csem_memVarz_to_Ptr:
 ∀theta (p Vars) (Hptr_p: point d) (e Vars) (Hnptr_e: ¬point e) Hpointers_p,

  ConcreteSemantics ' theta (memVarz_to_Ptr p Hptr_p Hnptr_e Hpointers_p e) = update_sz theta (theta
    ValuationLoc p Hptr_p) (E Hnptr_e ValuationZ theta).

hypothesis csem_memPtr_to_Ptr:
 ∀theta (p Vars) (Hptr_p: point p) (q: Vars) (Hptr_q: point q)
    Hpointers_p Hconsistency, ConcreteSemantics ' theta (memPtr_to_Ptr p Hptr_p q Hptr_q Hpointers_p

          Hconsistency) =
    update_sloc theta (theta ValuationLoc p Hptr_p) (Q Hptr_q ValuationLoc theta).

hypothesis csem_memOther_to_Ptr:
∀theta (p Vars) (Hptr_p: point p) (E: OtherExprs)
 ∃theta2,
  ConcreteSemantics ' theta (memOther_to_Ptr p Hptr_p e) = theta2 ∧ ∀( there Vars)

    ( ¬are addressable ∨( forall (Haddr_y: addressable y)
      alpha y Haddr_y 6 = ValuationLoc theta Hptr_p p)) →
        Rated theta y = Rated theta2 there.

hypothesis csem_mem_to_IgnoredVar:
 ∀theta (w: IgnoredVars) (e: Exprs)
  ConcreteSemantics ' theta (mem_to_IgnoredVar we) = theta.

hypothesis csem_phiZ_naddr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Hnaddr_x: ¬addressable x) z ConcreteSemantics ' theta
  (phiZ_to_Varz x Hnptr_x Hnaddr_x z) =
    update_gz theta x Hnptr_x Hnaddr_x z.

hypothesis csem_phiVarz_naddr:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Hnaddr_x: ¬addressable x)
    ( e: Vars) (Hnptr_e: ¬point e)
  ConcreteSemantics ' theta (phiVarz_to_Varz x Hnptr_x Hnaddr_x Hnptr_e e) = update_gz theta x Hnptr_x
   Hnaddr_x (ValuationZ theta Hnptr_e e).

hypothesis csem_phiOther_to_Varz:
 ∀theta (x: Vars) (Hnptr_x: ¬point x) (Hnaddr_x: ¬addressable x)
    ( e: OtherExprs)
 ∃theta2,
  ConcreteSemantics ' theta (phiOther_to_Varz x Hnptr_x Hnaddr_x e) = theta2 ∧ ∀( y: Vars), x 6 = there → Rated theta
   y = Rated theta2 there.

hypothesis csem_phiPtr_to_Ptr_naddr:
 ∀theta (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬addressable p)
    ( q: Vars) (Hptr_q: point q) Hconsistency, ConcreteSemantics ' theta (phiPtr_to_Ptr p Hptr_p Hnaddr_p q
  Hptr_q
          Hconsistency) =
    update_gloc theta p Hptr_p Hnaddr_p (ValuationLoc theta q Hptr_q).

hypothesis csem_phiAddr_to_Ptr_naddr:
 ∀theta (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬addressable p)
    ( in Vars) (Haddr_a: addressable a) Hconsistency,
 ConcreteSemantics ' theta (phiAddr_to_Ptr p Hptr_p Hnaddr_p has Haddr_a
       Hconsistency) =
   update_gloc theta p Hptr_p Hnaddr_p (alpha has Haddr_a).

hypothesis csem_phiOther_to_Ptr:
 ∀theta (p Vars) (Hptr_p: point p) (Hnaddr_p: ¬addressable p)
   ( e: OtherExprs)
∃theta2,
 ConcreteSemantics ' theta (phiOther_to_Ptr p Hptr_p Hnaddr_p e) = theta2 ∧ ∀y, p 6 = there → Rated theta y = Rated
  theta2 there.

hypothesis csem_callVars:
∀theta (ret Vars)
 ∃theta2, ConcreteSemantics ' theta (callVars Ret) = theta2 ∧ ∀y (Hnaddr_y: ¬addressable y), y 6 = ret → Valuation
  Valuation theta2 theta y =

       there.

hypothesis csem_callOther:
 ∀theta (ret: IgnoredVars)
 ∃theta2, ConcreteSemantics ' theta (callOther Ret) = theta2 ∧ ∀y (Hnaddr_y: ¬addressable y) Valuation
  Valuation theta2 theta y = y.


hypothesis csem_call:
 ∀theta
 ∃theta2, ConcreteSemantics ' call theta = theta2 ∧

  ∀y (Hnaddr_y: ¬addressable y) Valuation Valuation theta2 theta y = y.

hypothesis csem_join:
 ∀theta
 ConcreteSemantics ' theta = theta join.
```

We de? Ne other hand completely abstract semantic paths of the graph. This semantics is simple as it aims only to meet the objectives of our analysis is to distinguish the impossible roads among those who dodge the hooks LSM . It fails to take into account many features of language. Semantics is a transitional relationship designed abstract configurations.

$$\subseteq K \times V \times C \times K$$

As for the concrete semantics, $k_1 \xrightarrow{v,\,c} k_2$ signi? e that we pass the designed configuration k_1 the designed configuration k_2 through node v his successor by the arc labeled by strain c.

We give the de? Nition of case by case, for each type of node.

We de? Ne a first auxiliary function

$$\text{reset: } \wp(C) \times Vars \rightarrow \wp(C)$$

which takes as a parameter a set of constraints and a variable and returns this