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

Laurent GEORGET*
Mathieu JAUME†  Guillaume PIOLLE‡
Frédéric TRONEL‡  Valérie VIET TRIEM TONG‡

*Université de Rennes 1 / ‡CentraleSupélec / Inria, Rennes, France
†LIP6, Sorbonne Universités, Paris, France
An Information Flow Perspective

- **Linux** Operating Systems

  **Containers of information**: objects in the system able to store information originating from users, the OS environment, etc.:
  - files
  - pipes
  - network sockets
  - message queues
  - processes’ memory space
  - more?

**Data flow** from one container to another
  - when reading a file
  - when storing a message in a message queue
  - etc.
The information must flow

- Process
- Userspace
- Syscall interface
- Kernelspace
- Kernel thread

Hardware

- User processes are isolated
- Have no privileges
- Must use **System Calls** to perform privileged operations
The information must flow

- Userspace
  - Process
  - User processes are isolated
  - Have no privileges
  - Must use **System Calls** to perform privileged operations

- Kernelspace
  - Kernel thread

- Hardware

- Syscalls cause information flows
Information Flow Trackers for Linux

- **Laminar** Porter et al., “Practical Fine-Grained Information Flow Control Using Laminar”

- **KBlare** Zimmermann, Mé, and Bidan, “An Improved Reference Flow Control Model for Policy-Based Intrusion Detection”

- **Weir** Nadkarni et al., “Practical DIFC enforcement on Android”
TRACKING FLOWS WITH TAIN'T PROPAGATION

- Each container has a label identifying its initial content
- Each time a flow occurs, the destination label is updated with the source label

Example: head file | wc

```
/bin/head
/sh
/file
/bin/wc
```
TRACKING FLOWS WITH TAINT PROPAGATION

➤ Each container has a **label** identifying its initial content
➤ Each time a flow occurs, the destination label is **updated** with the source label
➤ Example: `head file | wc`
Tracking flows with taint propagation

- Each container has a **label** identifying its initial content
- Each time a flow occurs, the destination label is **updated** with the source label

**Example:** head file | wc
Each container has a **label** identifying its initial content

Each time a flow occurs, the destination label is **updated** with the source label

**Example:** head file | wc
Each container has a **label** identifying its initial content.

Each time a flow occurs, the destination label is **updated** with the source label.

Example: `head file | wc`
Tracking flows with taint propagation

- Each container has a label identifying its initial content
- Each time a flow occurs, the destination label is updated with the source label

Example: `head file | wc`
TRACKING FLOWS WITH TAINT PROPAGATION

- Each container has a **label** identifying its initial content
- Each time a flow occurs, the destination label is **updated** with the source label

**Example:** `head file | wc`
Example 1: read

fs/read_write.c
Graphs and execution paths

- One system call = One graph
- One possible execution path = One path from INIT to END
- One instruction = One node
- One sequence or jump = One edge

Extracted directly from the GCC compiler

Not exactly C but GIMPLE: intermediate representation

In Static Single Assignment form

---

¹Cytron et al., “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”.
/*
 * This routine simulates a hangup
 * on the tty, to arrange that
 * users are given clean terminals
 * at login time.
 */
SYSCALL_DEFINE0(vhangup)
{
    if (capable(CAP_SYS_TTY_CONFIG)) {
        tty_vhangup_self();
        return 0;
    }
    return -EPERM;
}
ANATOMY OF A SYSCALL

Syscall = Entry-point of a user process in the kernel
**Anatomy of a syscall**

Syscall = Entry-point of a user process in the kernel

1. Basic checks
2. Advanced checks / lock taking
3. Linux Security Modules hooks
4. Actual operation
5. Lock release
6. Return
**Anatomy of a syscall**

Syscall = Entry-point of a user process in the kernel

1. Basic checks
2. Advanced checks / lock taking
3. **Linux Security Modules hooks**
4. Actual operation
5. Lock release
6. Return

Many shortcuts exist, in case of errors.
THE **Linux Security Modules Framework**

LSM provides security kernel developers with:

- Additional general-purpose **security fields** in kernel data structures (inodes, tasks, etc.)
- **Hooks** strategically placed in the sysscalls code to register callbacks
THE LINUX SECURITY MODULES FRAMEWORK

LSM provides security kernel developers with:

▶ Additional general-purpose security fields in kernel data structures (inodes, tasks, etc.)

▶ Hooks strategically placed in the syscalls code to register callbacks

▶ Expected use: LSMs store their state in the fields and use the hooks to
  ▶ manage the state
  ▶ authorize security-sensitive operations
Our problem

Information flow trackers can only observe the execution of syscalls when called through a LSM hook.

If a syscall can generate an information flow without going through a LSM hook, that flow will be missed.
Our problem

Information flow trackers can only observe the execution of syscalls when called through a LSM hook.

If a syscall can generate an information flow without going through a LSM hook, that flow will be missed.

Important property to ensure a correct flow tracking

There must be a LSM hook in each execution path leading to the production of a flow in system calls.
Previous works


- Jaeger, Edwards, and Zhang, “Consistency analysis of authorization hook placement in the Linux security modules framework”


- Muthukumaran, Jaeger, and Ganapathy, “Leveraging "choice" to automate authorization hook placement”
Finding problematic paths
FINDING PROBLEMATIC PATHS
Finding problematic paths
Finding problematic paths
**Finding problematic paths**

\[ P \] is the set of apparently valid paths generating flows **not** covered by a LSM hooks → paths to analyze
Instructions causing flows and LSM hooks

LSM hooks can be automatically found in the code of system calls.
Instructions causing flows less so…
Instructions causing flows and LSM hooks

LSM hooks can be automatically found in the code of system calls

Instructions causing flows less so…

Several heuristics:

▶ Use of locking
▶ End of checks
▶ Calls to architecture/hardware-dependent functions
▶ Dynamic calls through function pointers
Several standard problems

Some paths are actually **impossible**: we should exclude them

**Loops** mean there are an infinity of paths of finite length: we cannot analyze them all
The complete mediation holds if, and only if: $P \subseteq I$, i.e. all the execution paths that perform an information flow and are not controlled by the information flow monitor since they do not contain an LSM hook are impossible according to the static analysis.
Dealing with impossible paths and loops

Property (Complete mediation)

The complete mediation holds if, and only if: \( P \subseteq I \), i.e. all the execution paths that perform an information flow and are not controlled by the information flow monitor since they do not contain a LSM hook are impossible according to the static analysis.
Dealing with impossible paths and loops

Property (Complete mediation)

The complete mediation holds if, and only if: \( P \subseteq I \), i.e. all the execution paths that perform an information flow and are not controlled by the information flow monitor since they do not contain a LSM hook are impossible according to the static analysis.
Since \( \mathbf{P} \) may be infinite, we need a way to make the analysis of the subset of acyclic paths in \( \mathbf{P} \) sufficient to conclude on all paths in \( \mathbf{P} \).
**Analysis outline**

General idea:

- Analyze each system call independently
- In each system call,
  1. identify nodes producing flows
  2. trace the paths back up until reaching either the beginning of the function or a LSM hook
  3. discard the paths reaching a LSM hook (paths in $Paths_{LSM}$)
  4. when reaching a loop, jump to the outer-most loop header to select only acyclic paths
- For each analyzed path,
  - go through each node and edge in order
  - gather constraints on variables from nodes and guards on edges in a configuration
  - when reaching a configuration with inconsistent constraints, declare the path as impossible
  OR when reaching the end of the path, declare it as possible
Satisfiability

Current node: \( a.1 = \text{PHI}<0, a.8> \)

Set of constraints:
\[ \{ a.1 = 0 \} \]

Satisfiable: Yes
Satisfiability
Current node: <ssa 1>.6 = f(a.1)
Set of constraints: \{a.1 = 0\}
Satisfiable: Yes
Satisfiability

Current edge: \[ !<\text{ssa } 1>.6 \neq a.1 \]

Set of constraints:
\[
\begin{cases}
  a.1 = 0, \\
  <\text{ssa } 1>.6 \neq a.1
\end{cases}
\]

Satisfiable: Yes
Satisfiability

Current node: \( a.11 = \text{PHI}<a.1> \)

Set of constraints:

\[
\begin{align*}
& a.1 = 0, \\
& <ssa~1>.6 \neq a.1, \\
& a.11 = a.1
\end{align*}
\]

Satisfiable: Yes
Satisfiability

Current edge: \([a.11 > 10]\)

Set of constraints:
\[
\begin{cases} 
    a.1 = 0, \\
    \langle ssa 1 \rangle.6 \neq a.1, \\
    a.11 = a.1, \\
    a.11 > 10 
\end{cases}
\]

Satisfiable: No \implies path impossible
**Satisfiability**

Set of constraints:

\[
\begin{cases}
\ a.1 = 0, \\
\ <ssa\ 1>.6 \neq a.1, \\
\ a.11 = a.1, \\
\ a.11 > 10
\end{cases}
\]

Satisfiable: \textbf{No} \implies \text{path impossible}

The satisfiability is verified by SMT-solver Yices\(^2\).

Handling loops

Dealing with loops

Loops have a special syntax and are detected by GCC.

We define an equivalence relation on paths: two paths are equivalent if they are identical up to their cycles.

We analyze only acyclic paths (normal form).

When there is a loop, we remove constraints about all variables modified inside the loop. →

The number of iterations of loops does not change the resulting configuration.
IMPLEMENTATION

The analysis is implemented as Kayrebt::PathExaminer2, a GCC 4.8 plugin\(^3,4\).

No extraction of CFGs needed: the analysis works on GCC’s CFG.

Deep insertion inside the compilation process: after the optimized phase.

Needs a previous annotation of nodes causing information flows and inlinable functions (can be done with Kayrebt::Callgraphs)


\(^4\)Emese Revfy. Introduce GCC plugin infrastructure. Published: Patch submitted to the kernel mailing-list. 2016.
Results — Explanations

✓ : Everything is alright, complete mediation is ensured

∼ : We have identified some problems: some paths which should be impossible and are not

× : We wanted to analyze the paths but there are actually no LSM hooks in the system call
## Results — read, write, AND THEIR KIN

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>read......</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>readv.....</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>preadv....</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>pread64...</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>write.....</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>writev....</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>pwritev...</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>pwrite64..</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>sendfile..</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
<tr>
<td>sendfile64</td>
<td>✓</td>
<td>All paths in P are impossible</td>
</tr>
</tbody>
</table>
## Results — splice-like system calls

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>splice..</td>
<td>~</td>
<td>No hook for the pipe-to-pipe flow</td>
</tr>
<tr>
<td>..</td>
<td>×</td>
<td>All other paths are impossible</td>
</tr>
<tr>
<td>tee......</td>
<td>×</td>
<td>No LSM hook</td>
</tr>
<tr>
<td>vmsplice</td>
<td>~</td>
<td>One path is possible</td>
</tr>
</tbody>
</table>
## Results — Network-Specific System Calls

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>recv ....</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>recvmsg.</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>recvmmsg</td>
<td>~</td>
<td>One path is possible</td>
</tr>
<tr>
<td>recvfrom</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>send ....</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>sendmsg.</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>sendmmsg</td>
<td>~</td>
<td>One path is possible</td>
</tr>
<tr>
<td>sendto ..</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
</tbody>
</table>
# Results — processes’ life

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>✓</td>
<td>Set P is empty</td>
</tr>
<tr>
<td>vfork</td>
<td>✓</td>
<td>Set P is empty</td>
</tr>
<tr>
<td>clone</td>
<td>✓</td>
<td>Set P is empty</td>
</tr>
<tr>
<td>execve</td>
<td>✓</td>
<td>Set P is empty</td>
</tr>
<tr>
<td>execveat</td>
<td>✓</td>
<td>Set P is empty</td>
</tr>
</tbody>
</table>
# Results — System V and POSIX message queues

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgrcv</td>
<td>✓</td>
<td>All paths in $P$ are impossible</td>
</tr>
<tr>
<td>msgsnd</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>mq_timedreceive</td>
<td>✗</td>
<td>No LSM hook</td>
</tr>
<tr>
<td>mq_timedsend...</td>
<td>✗</td>
<td>No LSM hook</td>
</tr>
</tbody>
</table>
# Results — Memory-to-memory flows

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Result</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>process_vm_readv</td>
<td>✓</td>
<td>Some paths possible but not considered an actual flow</td>
</tr>
<tr>
<td>process_vm_writev</td>
<td>✓</td>
<td>Some paths possible but not considered an actual flow</td>
</tr>
<tr>
<td>migrate_pages</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>move_pages</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>shmat</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>mmap_pgoff</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>mmap</td>
<td>✓</td>
<td>Set $P$ is empty</td>
</tr>
<tr>
<td>ptrace</td>
<td>✓</td>
<td>Some paths possible but not considered an actual flow</td>
</tr>
</tbody>
</table>
Interesting results:

- confort the idea that it is possible to do information flow tracking with LSM
- highlight some holes in the design and implementation of LSM with respect to information flow tracking
- give a verifiable and reproducible way to analyze and improve the LSM framework
Static analysis assisted by the compiler

The GCC plugin interface has been opened to implement optimizations passes.

But! It is also a new way of performing static analysis!

Already used in the Linux kernel\(^5\)

STATIC ANALYSIS ASSISTED BY THE COMPILER

The GCC plugin interface has been opened to implement optimizations passes.

But! It is also a new way of performing static analysis!

Already used in the Linux kernel\(^5\)

Benefits

- GCC data structures available: CFGs, points-to oracle, etc.
- Analysis can be done on simpler intermediate representations
- Ability to deal with GCCisms
- The code that is analyzed is not the code that is written but the code that will get **executed** (or at least, a closer form thereof)

---

**On-going work**

Cover more overt and covert channels of information flows in a correct, verifiable way.

In particular, deal with mmap-ed files and shared memories.

Deal with concurrency between flows.
Thank you for your attention.

Questions?


Variables are separated in 2x2 categories:

- $\mathcal{V}_{\text{mem}}$ vs. $\mathcal{V}_{\text{temp}}$
  - $\mathcal{V}_{\text{mem}}$: Aliasable variables
  - $\mathcal{V}_{\text{temp}}$: Variables whose address is never taken

- $\mathcal{V}_{\text{ptr}}$ vs. $\mathcal{V}_{\mathbb{Z}}$
  - $\mathcal{V}_{\text{ptr}}$: Pointers
  - $\mathcal{V}_{\mathbb{Z}}$: Numeric variables

The typing is enforced by the compiler.

Many variables are synthetized by the compiler itself to maintain the SSA property.
Node types

Simple assignments

\[ \text{Case } x = y \quad \text{Add a constraint } x = y \]
\[ \text{Case } p = &y \quad \text{Add a mapping } p \leftrightarrow y \]
Assignments through pointers

\[ *a.1 = y \]

Effects:

- If there is a mapping \( a.1 \leftrightarrow x \), add a constraint \( x = y \)
- Otherwise, remove all constraints about variables \( a.1 \) may point to (GCC has a points-to oracle)
**Node types**

**Phi nodes**

\[
<\text{ssa 184}.88 = \text{PHI<ssa 183>.87, retval.83}>
\]

Found after nodes where several edges meet.

Effects:

\[
x = \text{PHI} < e_1, e_2, \ldots, e_n >
\]

Add a constraint \( x = e_i \) where \( e_i \) correspond to the branch taken in this path
Function calls

\[
\text{retval.85} = \text{security_file_permission(file.7, 4)}
\]

Effects:
- Remove constraints on the return value
- Remove constraints on variables in $\mathcal{V}_{\text{ars}}^{\text{mem}}$

Portions of assembly code are also represented with this node.
Effects:

- Add the constraint corresponding to the guard
- The operator is one of \{=, \neq, <, >, \geq, \leq\}

Guards on edges with the same source node are complementary