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

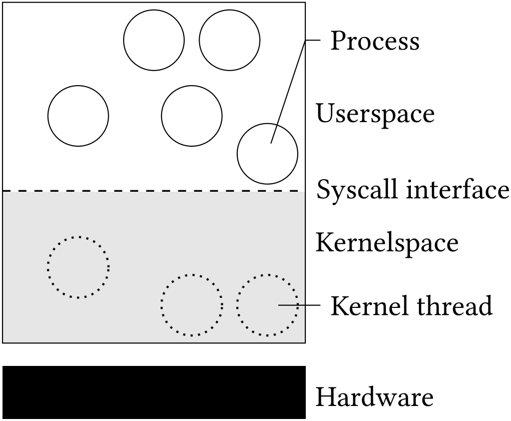
Containers of information: objects in the system able to store information originating from users, the OS environment, etc.:

- files
- pipes
- network sockets
- message queues
- processes' memory space
- more?

Data **flow** from one container to another

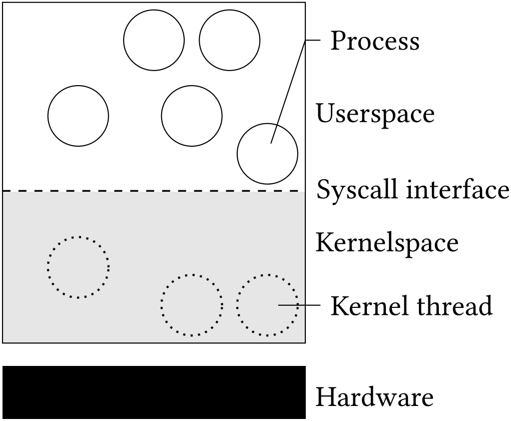
- when reading a file
- when storing a message in a message queue
- etc.

The information must flow



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

The information must flow



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

- Syscalls cause information flows

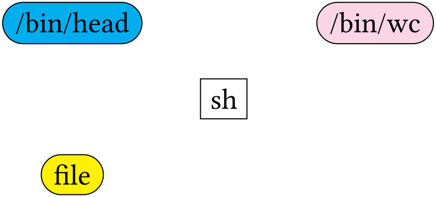
Information Flow Trackers for Linux

- **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 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

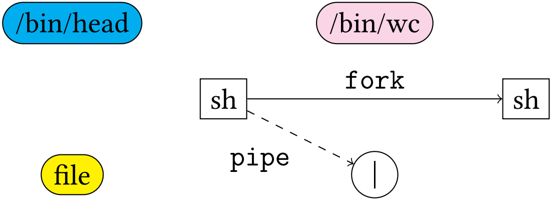
- Example: `head file | wc`



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

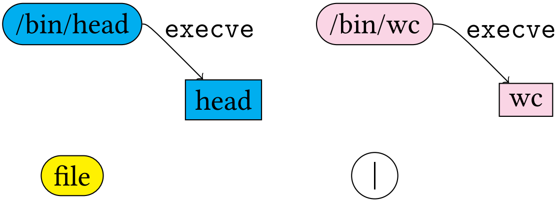
- Example: `head file | wc`



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

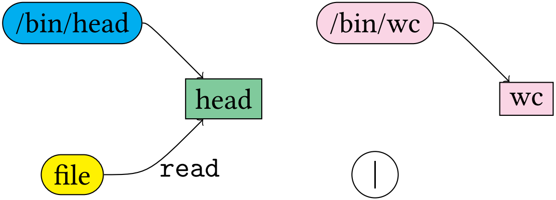
- Example: `head file | wc`



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

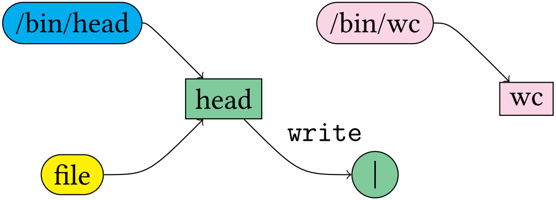
- Example: `head file | wc`



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

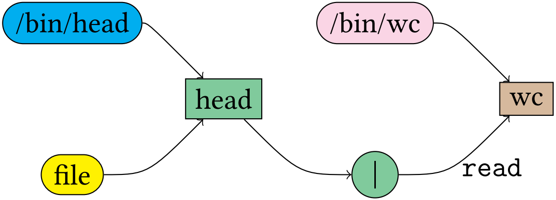
- Example: `head file | wc`



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

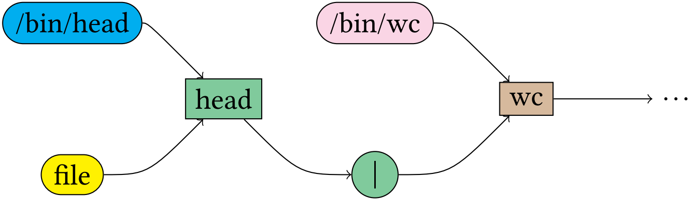
- Example: `head file | wc`



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

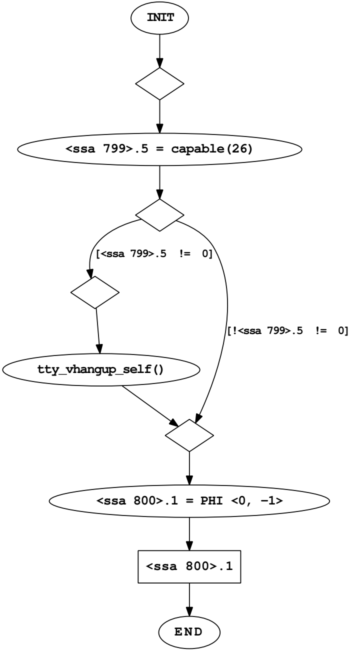
- One system call = One graph
- One possible execution path = One path from `INIT` to `END`
- One instruction = One node
- One sequence or jump = One edge

Extracted directly from the **GCC compiler**

Not exactly C but **GIMPLE**: intermediate representation

In *Static Single Assignment* form[1]

[1] Cytron 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 developpers with:

- Additional general-purpose **security fields** in kernel data structures (inodes, tasks, etc.)
- **Hooks** strategically placed in the syscalls code to register callbacks

The *Linux Security Modules* Framework

LSM provides security kernel developpers 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.

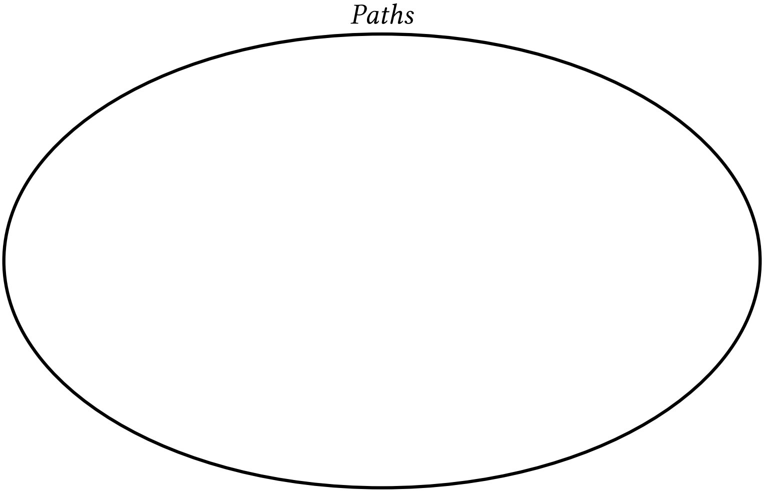
Important property to ensure a correct flow tracking

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

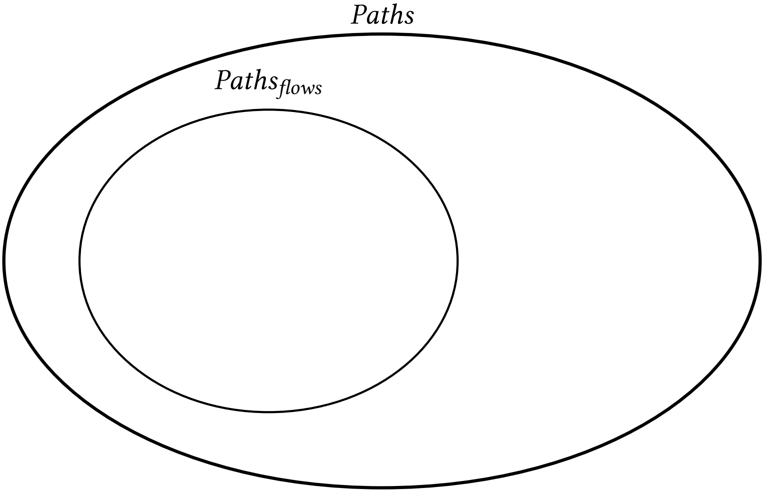
Previous works

- Zhang, Edwards, and Jaeger, “Using CQUAL for Static Analysis of Authorization Hook Placement”
- Jaeger, Edwards, and Zhang, “Consistency analysis of authorization hook placement in the Linux security modules framework”
- Ganapathy, Jaeger, and Jha, “Automatic Placement of Authorization Hooks in the Linux Security Modules Framework”
- Muthukumaran, Jaeger, and Ganapathy, “Leveraging "choice" to automate authorization hook placement”

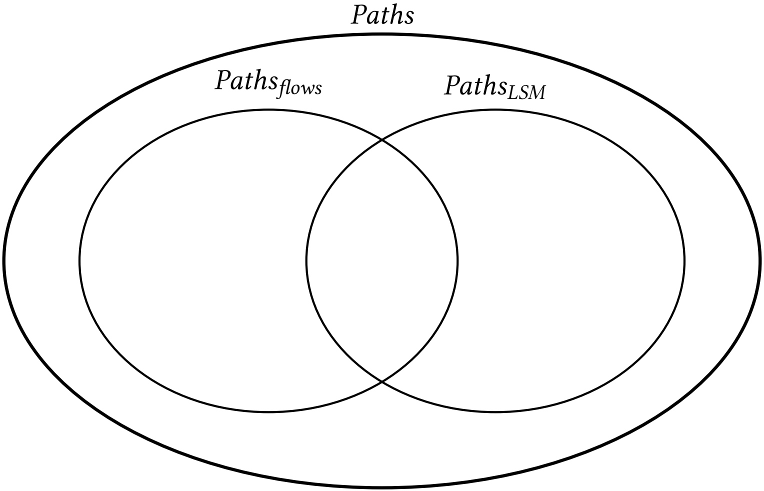
Finding problematic paths



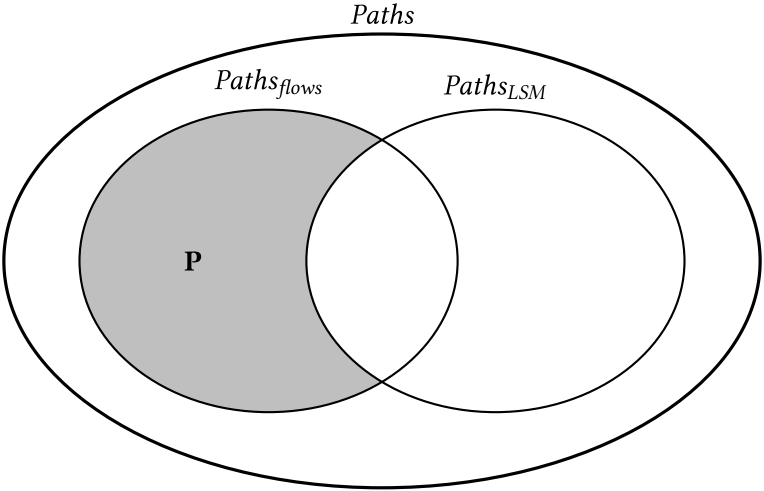
Finding problematic paths



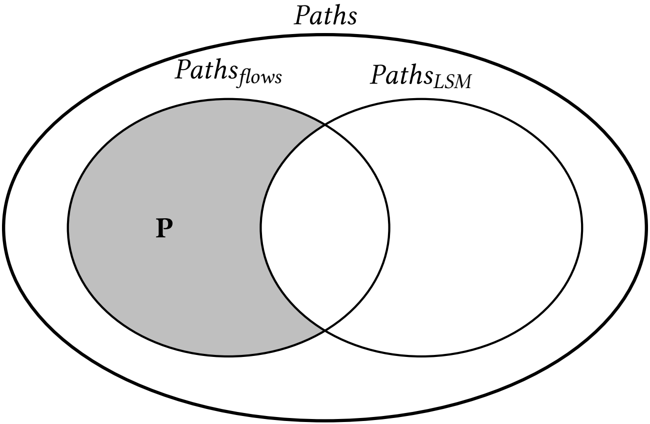
Finding problematic paths



Finding problematic paths



Finding problematic paths



P is the set of apparently valid paths generating flows **not** covered by a LSM hooks $\implies$ 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…

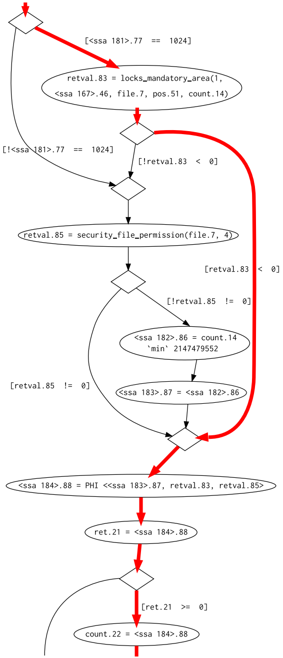
Instructions causing flows and LSM hooks

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

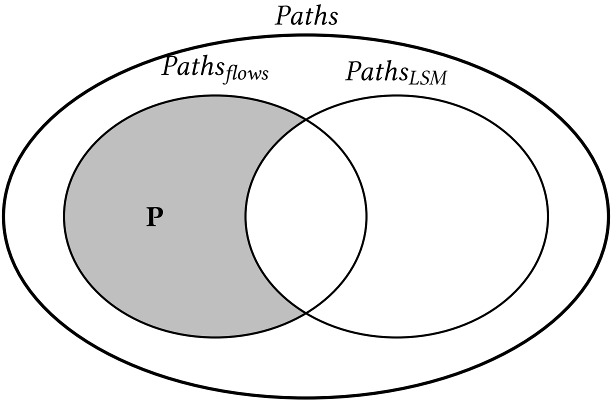

[<ssa 181>.77 == 1024]
retval.83 = locks_mandatory_area(1, <ssa 167>.46, file.7, pos.51, count.14)
[!<ssa 181>.77 == 1024]
[!retval.83 < 0]
retval.85 = security_file_permission(file.7, 4)
[retval.83 < 0]
[!retval.85 != 0]
<ssa 182>.86 = count.14 `min` 2147479552
[retval.85 != 0]
<ssa 183>.87 = <ssa 182>.86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

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

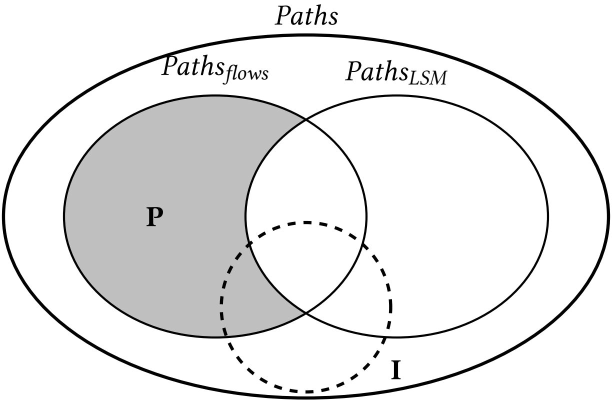
Dealing with impossible paths and loops



Property (Complete mediation)

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

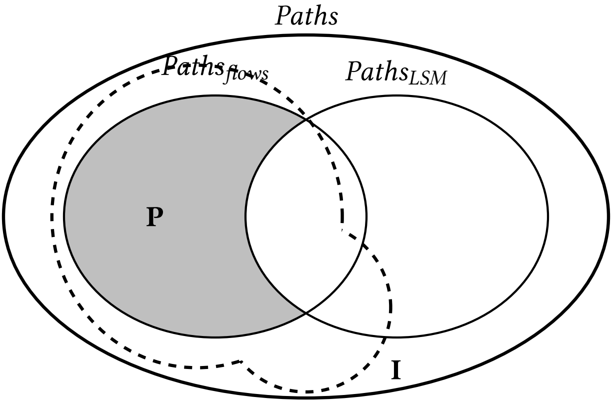
Dealing with impossible paths and loops



Property (Complete mediation)

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

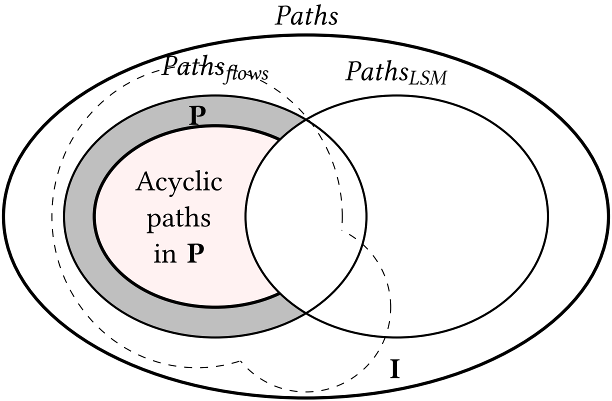
Dealing with impossible paths and loops



Property (Complete mediation)

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

Dealing with impossible paths and loops



- ▶ Since **P** may be infinite, we need a way to make the analysis of the subset of acyclic paths in **P** sufficient to conclude on all paths in **P**.

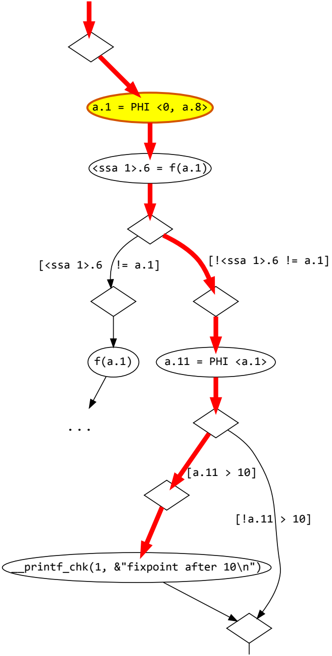
Analysis outline

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



Satisfiability

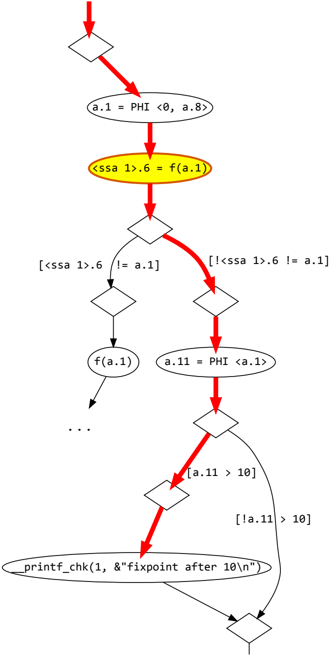
Current node: `a.1 = PHI<0,a.8>`

Set of constraints:

$\{a.1 = 0\}$

Satisfiable: Yes

Satisfiability



Satisfiability

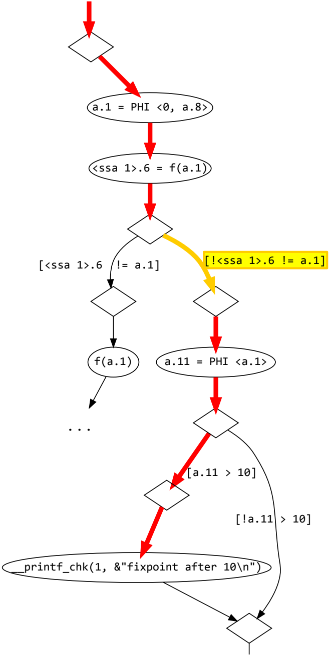
Current node: `<ssa 1>.6 = f(a.1)`

Set of constraints:

$\{a.1 = 0\}$

Satisfiable: Yes

Satisfiability



Satisfiability

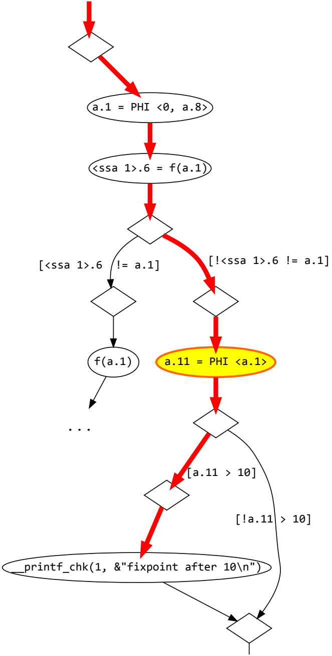
Current edge: `[!<ssa 1>.6 != a.1]`

Set of constraints:

$$\left\{\begin{array}{r} a.1 = 0, \\ \texttt{<ssa 1>}.6 \neq a.1 \end{array}\right\}$$

Satisfiable: Yes

Satisfiability



Satisfiability

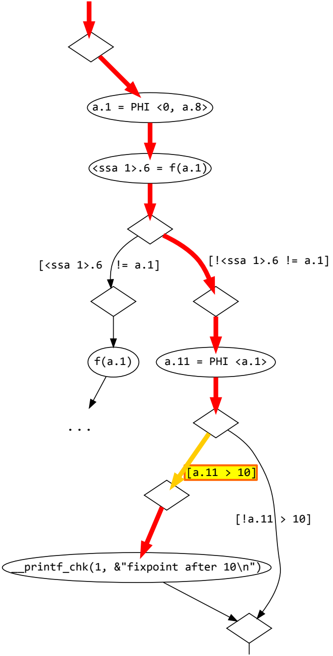
Current node: `a.11 = PHI<a.1>`

Set of constraints:

$$\left\{ \begin{array}{r} a.1 = 0, \\ \texttt{<ssa 1>}.6 \neq a.1, \\ a.11 = a.1 \end{array} \right\}$$

Satisfiable: Yes

Satisfiability



Satisfiability

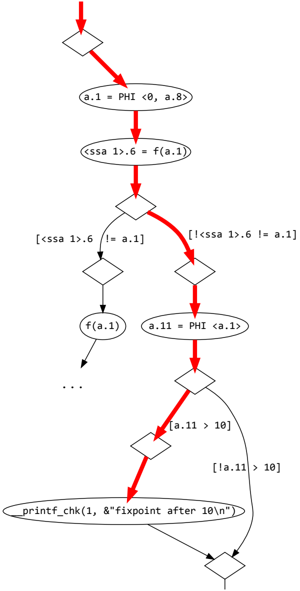
Current edge: `[a.11 > 10]`

Set of constraints:

$$\left\{ \begin{array}{r} a.1 = 0, \\ \texttt{<ssa 1>}.6 \neq a.1, \\ a.11 = a.1, \\ a.11 > 10 \end{array} \right\}$$

Satisfiable: **No** $\Longrightarrow$ path impossible

Satisfiability



Satisfiability

Set of constraints:

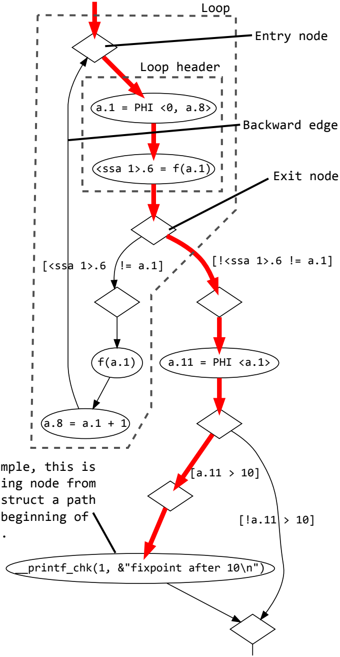
$$\left\{ \begin{array}{r} a.1 = 0, \\ \texttt{<ssa 1>}.6 \neq a.1, \\ a.11 = a.1, \\ a.11 > 10 \end{array} \right\}$$

Satisfiable: **No** $\Longrightarrow$ path impossible

The satisfiability is verified by SMT-solver Yices [2].

[2] Bruno Dutertre and Leonardo de Moura. **The Yices SMT solver**. Tech. rep. SRI International, 2006.

Handling loops



For this example, this is the interesting node from which we construct a path towards the beginning of the function.

Dealing with loops

Loops have a special syntax and are detected by GCC

We define a **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. $\Longrightarrow$ 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**)

[3] Richard Matthew Stallman and the GCC developer community. **Using the GNU Compiler Collection (GCC)**. . Tech. rep. 2013. URL: `https://gcc.gnu.org/onlinedocs/gcc-4.8.4/gcc/` (visited on 05/18/2015).

[4] Emese Revfy. **Introduce GCC plugin infrastructure**. Published: Patch submitted to the kernel mailing-list. 2016.

Results — Explanations

- ✓ : Everything is allright, 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

Syscall	Result	Details
`read`	✓	All paths in **P** are impossible
`readv`	✓	All paths in **P** are impossible
`preadv`	✓	All paths in **P** are impossible
`pread64`	✓	All paths in **P** are impossible
`write`	✓	All paths in **P** are impossible
`writev`	✓	All paths in **P** are impossible
`pwritev`	✓	All paths in **P** are impossible
`pwrite64`	✓	All paths in **P** are impossible
`sendfile`	✓	All paths in **P** are impossible
`sendfile64`	✓	All paths in **P** are impossible

Results — `splice`-like system calls

Syscall	Result	Details
`splice..`	∼	No hook for the pipe-to-pipe flow
`..`		All other paths are impossible
`tee.....`	×	No LSM hook
`vmsplice`	∼	One path is possible

RESULTS — NETWORK-SPECIFIC SYSTEM CALLS

Syscall	Result	Details
`recv....`	✓	Set **P** is empty
`recvmsg.`	✓	Set **P** is empty
`recvmmsg`	~	One path is possible
`recvfrom`	✓	Set **P** is empty
`send....`	✓	Set **P** is empty
`sendmsg.`	✓	Set **P** is empty
`sendmmsg`	~	One path is possible
`sendto..`	✓	Set **P** is empty

Results — processes' life

Syscall	Result	Details
`fork`	✓	Set **P** is empty
`vfork` ...	✓	Set **P** is empty
`clone` ...	✓	Set **P** is empty
`execve` ..	✓	Set **P** is empty
`execveat`	✓	Set **P** is empty

Results — System V and POSIX message queues

Syscall	Result	Details
`msgrcv`	✓	All paths in **P** are impossible
`msgsnd`	✓	Set **P** is empty
`mq_timedreceive`	×	No LSM hook
`mq_timedsend` ...	×	No LSM hook

Results — Memory-to-memory flows

Syscall	Result	Details
`process_vm_readv`	✓	Some paths possible but not considered an actual flow
`process_vm_writev`	✓	Some paths possible but not considered an actual flow
`migrate_pages`	✓	Set **P** is empty
`move_pages`	✓	Set **P** is empty
`shmat`	✓	Set **P** is empty
`mmap_pgoff`	✓	Set **P** is empty
`mmap`	✓	Set **P** is empty
`ptrace`	✓	Some paths possible but not considered an actual flow

Outcome

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]

[5] Emese Revfy. **Introduce GCC plugin infrastructure**. Published: Patch submitted to the kernel mailing-list. 2016.

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)

[5] Emese Revfy. **Introduce GCC plugin infrastructure**. Published: Patch submitted to the kernel mailing-list. 2016.

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?

Ron Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: **ACM Transactions on Programming Languages and Systems** 13.4 (Oct. 1991), pp. 451–490. URL: `http://doi.acm.org/10.1145/115372.115320`.

Bruno Dutertre and Leonardo de Moura. **The Yices SMT solver**. Tech. rep. SRI International, 2006.

Vinod Ganapathy, Trent Jaeger, and Somesh Jha. “Automatic Placement of Authorization Hooks in the Linux Security Modules Framework”. In: **ACM Conference on Computer and Communications Security**. ACM, 2005.

Trent Jaeger, Antony Edwards, and Xiaolan Zhang. “Consistency analysis of authorization hook placement in the Linux security modules framework”. In: **ACM Trans. Inf. Syst. Secur.** 7.2 (2004).

Divya Muthukumaran, Trent Jaeger, and Vinod Ganapathy. “Leveraging "choice" to automate authorization hook placement”. In: **ACM Conference on Computer and Communications Security**. ACM, 2012.

Adwait Nadkarni et al. “Practical DIFC enforcement on Android”. In: **USENIX Security Symposium**. 2016.

Donald E. Porter et al. “Practical Fine-Grained Information Flow Control Using Laminar”. In: **ACM Transactions on Programming Languages and Systems** 37.1 (Nov. 2014).

Emese Revfy. **Introduce GCC plugin infrastructure**. Published: Patch submitted to the kernel mailing-list. 2016.

Richard Matthew Stallman and the GCC developer community. **Using the GNU Compiler Collection (GCC)**. Tech. rep. 2013. URL: https://gcc.gnu.org/onlinedocs/gcc-4.8.4/gcc/ (visited on 05/18/2015).

- Xiaolan Zhang, Antony Edwards, and Trent Jaeger. “Using CQUAL for Static Analysis of Authorization Hook Placement”. In: **USENIX Security Symposium**. USENIX Association, 2002.
- Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. “An Improved Reference Flow Control Model for Policy-Based Intrusion Detection”. In: **Computer Security – ESORICS 2003**. Lecture Notes in Computer Science 2808. Springer Berlin Heidelberg, Oct. 13, 2003, pp. 291–308.

Variables

Variables are separated in 2x2 categories:

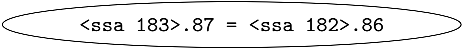
- $\mathcal{V}ars^{mem}$ vs. $\mathcal{V}ars^{temp}$
  - $\mathcal{V}ars^{mem}$: Aliasable variables
  - $\mathcal{V}ars^{temp}$: Variables whose address is never taken
- $\mathcal{V}ars^{ptr}$ vs. $\mathcal{V}ars^{\mathbb{Z}}$
  - $\mathcal{V}ars^{ptr}$: Pointers
  - $\mathcal{V}ars^{\mathbb{Z}}$: Numeric 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

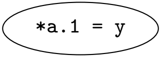


```
<ssa 183>.87 = <ssa 182>.86
```

Effects:	Case $x = y$	Add a constraint $x = y$
	Case $p = \& y$	Add a mapping $p \leftrightarrow y$

Node types

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

```
<ssa 184>.88 = PHI«ssa 183>.87, retval.83>
```

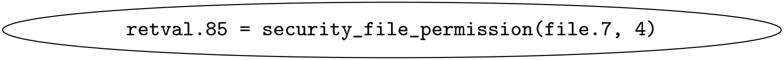
Found after nodes where several edges meet.

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

Node types

Function calls



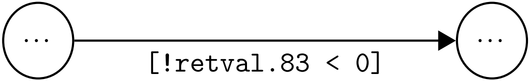
```
retval.85 = security_file_permission(file.7, 4)
```

Effects:

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

Portions of assembly code are also represented with this node

Edges



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