Specifying and Verifying Information Flow Control in SELinux Configurations

LORENZO CERAGIOLI, IMT School for Advanced Studies Lucca, Italy
LETTERIO GALLETTA, IMT School for Advanced Studies Lucca, Italy
PIERPAOLO DEGANO, Università di Pisa, Italy and IMT School for Advanced Studies Lucca, Italy
DAVID BASIN, ETH Zurich, Switzerland

Security Enhanced Linux (SELinux) is a security architecture for Linux implementing Mandatory Access Control. It has been used in numerous security-critical contexts ranging from servers to mobile devices. However, its application is challenging as SELinux security policies are difficult to write, understand, and maintain. Recently, the intermediate language CIL was introduced to foster the development of high-level policy languages and to write structured configurations. Despite CIL's high level features, CIL configurations are hard to understand as different constructs interact in non-trivial ways. Moreover, there is no mechanism to ensure that a given configuration obeys desired information flow policies. To remedy this, we enrich CIL with a formal semantics, and we propose IFCIL, a backward compatible extension of CIL for specifying fine-grained information flow requirements. Using IFCIL, administrators can express confidentiality, integrity, and non-interference properties. We also provide a tool to statically verify these requirements and we experimentally assess it on ten real-world policies.

CCS Concepts: • **Theory of computation → Verification by model checking**; • **Security and privacy → Logic and verification**.

Additional Key Words and Phrases: access control, information flow control

ACM Reference Format:
Lorenzo Ceragioli, Letterio Galletta, Pierpaolo Degano, and David Basin. 2024. Specifying and Verifying Information Flow Control in SELinux Configurations. *ACM Trans. Priv. Sec.* 1, 1 (August 2024), 38 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Security Enhanced Linux (SELinux) is a set of extensions of the Linux kernel that implements a Mandatory Access Control mechanism. It is widely used for defining security polices in Linux-based systems, including servers [50], network appliances [20], and mobile devices [47]. Defining an SELinux policy is conceptually simple: the system administrator defines a set of *types*, uses them to label all system resources and processes, and then defines a set of rules specifying which operations the processes can perform on resources. However, its use is far from simple. Writing, understanding, and maintaining SELinux security policies is difficult and error-prone as evidenced by numerous examples of misconfigurations [27] that have led to serious vulnerabilities in widely used policies.

Authors' Contact Information: Lorenzo Ceragioli, lorenzo.ceragioli@imtlucca.it, IMT School for Advanced Studies Lucca, Lucca, Italy; Letterio Galletta, letterio.galletta@imtlucca.it, IMT School for Advanced Studies Lucca, Lucca, Italy; Pierpaolo Degano, pierpaolo.degano@unipi.it, Università di Pisa, Pisa, Italy and IMT School for Advanced Studies Lucca, Lucca, Italy; David Basin, basin@inf.ethz.ch, ETH Zurich, Zurich, Switzerland.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2471-2574/2024/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To simplify working with SELinux and to address the limitations of its default policy language, the community called for and proposed new high-level configuration languages [26, 46]. In particular, SELinux developers recently proposed the intermediate configuration language CIL (Common Intermediate Language), which is a declarative language that offers advanced features to aid both policy specification and analysis. CIL supports the definition of structured configurations, using, e.g., namespaces and macros, and enables administrators to specify which resources are critical, which entities can access them, and which cannot. It also provides tool support to statically detect and prevent misconfigurations, which could lead to unauthorized access to security-critical resources.

However, CIL currently provides no means to prevent unwanted transitive information flows, which is essential to preventing confidentiality and integrity breaches. To overcome this serious limitation, we propose IFCIL, an extension of CIL supporting information flow requirements, and we endow it with a verification procedure for statically checking that a configuration satisfies its requirements.

Our proposal consists of three parts. First, we propose the domain specific language (DSL) called *IFL* (Information Flow Language), for expressing fine-grained information flow requirements, which we group in two categories: *functional* and *security* requirements. Functional requirements specify which permissions must be granted to users to perform their authorized tasks, such as which resources they can access and with which operations. In contrast, security requirements prevent entities from operating on other possibly critical entities, and thereby enforce security properties, including confidentiality, integrity, and non-transitive information flow properties. Our DSL is compositional and supports the refinement of requirements with further restrictions both to make them more demanding and to adapt them to specific contexts.

Second, we introduce *IFCIL* (Information Flow CIL), which extends CIL with constructs to annotate configurations with IFL requirements. Our extension is backward compatible: an IFCIL configuration is also a valid CIL configuration and can be translated by the standard CIL compiler.

Finally, we endow IFCIL with a verification procedure supported by an automated tool that, given a configuration, checks if its IFL requirements are satisfied. We assess our tool's effectiveness and scalability on numerous real-world configurations.

In summary, our main contributions are as follows.

- We present the language IFL for expressing complex, fine-grained, information flow requirements in a declarative and compositional way, including confidentiality, integrity, and non-transitive information flow properties. Various access control languages can easily be augmented with IFL. Moreover, IFL requirements can be verified using off-the-shelf LTL model checkers.
- We propose IFCIL, the integration of IFL inside CIL. We achieve this by using special comments that an administrator can associate with different parts of a CIL configuration. We give an algorithm for statically verifying the compliance of a configuration to its IFL requirements.
- We give CIL a formal semantics in a executable style, implement it in OCaml, and empirically validate its adequacy with respect to the CIL reference manual and the CIL compiler through systematic testing [9]. Besides providing the basis for our verification algorithm, the semantics and its experimental validation make it possible to understand CIL's trickier bits, and they illuminate unspecified corner cases and disagreements between the documentation and the compiler. We reported our findings to CIL's developers who fixed some of them in the compiler, and our semantics formalizes their new version. However, a corner case involving recursive definitions is still open (see Section 3).
- We provide a prototype tool [10] that implements our verification procedure by leveraging NuSMV, a popular model checker [12]. Our tool checks if an IFCIL configuration satisfies

the requirements therein and, when they are violated, it warns the administrator about potentially dangerous parts of the configuration.

- We experimentally assess our tool on ten real-world policies: Android base policies and vendor specific policies, SELinux base policies [21, 22], and a policy for OpenWRT routers [20]. We annotate these policies with IFL requirements expressing properties taken from the literature and with new ones. We thereby validate our tool and show that it scales well. For example, it takes less than two minutes to verify 39 requirements on the OpenWRT configuration, which has roughly 46,000 lines of code.

Outline: In Section 2 we introduce SELinux, CIL, and the mechanism used by administrators to protect critical resources. In Section 3 we give a high-level account of our CIL semantics and how we experimentally validate its adequacy. In Section 4 we present IFCIL and we explain our verification procedure for checking the satisfiability of the requirements in Section 5. In Section 6 we present our verification tool and our experimental assessment. In Section 7 we compare our work with the relevant literature and in Section 8 we discuss limitations and we draw conclusions. The appendices contain all the details of our formal development and proofs.

This is an extended version of our paper [11]. We present here a larger set of examples that better showcase some of CIL's intricacies. From a careful analysis of the examples, we derive some principles underlying CIL that lead to a simpler definition of its semantics. In doing so, we update our previous paper's semantics to formalize the changes that CIL's developers made to fix the bugs we have reported to them. We provide this new semantics in full detail and in an executable style, which is implemented in OCaml. We now extensively analyze and automatically validate the adequacy of our semantics using both randomized and hand-crafted tests. As examples of the problems we discovered we present three bugs in the CIL compiler, which have been fixed by the developers. Moreover, we discuss a case that we see as an inconsistency in the compiler, which has not been addressed. In addition to polishing the presentation of IFL and IFCIL, we introduce new IFCIL extensions to simplify writing information-flow requirements. We also report on improvements to our verification tool IFCILverif, and on a performance evaluation. Finally, we have used our tool to analyze seven new Android policies: five from the Android Open Source Project (OASP) and two that are vendor specific.

2 Background

SELinux. SELinux is a set of extensions to the Linux kernel and utilities [37]. It extends the major subsystems of the Linux kernel with strong, flexible, Mandatory Access Control. The SELinux security server permits or denies a process to invoke a system call on a resource based on a configuration specified by the system administrator. To specify a configuration, an administrator defines a set of *types*, and labels the OS resources and processes with them. In addition, all resources belong to predefined *classes*, such as file, process, socket, or directory. A rule in a configuration relates the type t and class c of resources, and the type t' of processes with the permitted operations. A rule thereby specifies the actions that processes labeled t' can perform on the resources of class c labeled t, for example, read or write a file, execute a process, open a socket, or change the Discretionary Access Control rights of a directory. A process P can invoke a system call on a resource only if there is a rule that permits P to do so.

Administrators typically specify configurations using SELinux's *kernel policy language* [39]. Configurations are then compiled to a kind of (kernel binary) access-control matrix. However, this policy language is very low-level. For example, it does not allow the administrator to structure configurations, which makes them hard to understand and maintain. Thus using the kernel policy language is cumbersome and error-prone, as shown by the over permissive evolution of the Android policy [27]. Some high-level configuration languages have been suggested with their own compilers and tools as an attempt to address these limitations [26, 33]. Recently, the SELinux developers proposed a promising new intermediate configuration language with advanced features and tools to support both the development of high-level languages and the definition of configurations. We briefly survey this language below.

CIL. The Common Intermediate Language (CIL) [38] was designed as a bridge between high-level configuration languages and the low-level binary representation introduced above. Compilers from various configuration languages to CIL are intended to support multi-language policy definitions. A compiler for the kernel policy language is currently available, and CIL is designed to support existing high-level configuration languages, e.g., Lobster [26], Cascade [4], and future ones too. Despite its original goal, CIL is also used to directly write configurations [20–22] for complex real-world policies, like for Android [19]. Indeed, CIL provides its users with high-level constructs like nested blocks, inheritance, and macros, thereby supporting the structured definition of configurations. Moreover, since CIL is declarative, it facilitates reasoning about configurations, and the same analysis techniques and tools for CIL can help when other high-level languages are used.

Roughly, a CIL configuration consists of a set of declarations of blocks, types, and rules. Similarly to classes in programming languages, *blocks* have names and introduce namespaces and further declarations. *Types* are labels that are associated with system resources and processes. Rules regulate types by specifying which operations processes can perform on resources. Intuitively, administrators can define two kinds of rules: those that grant permission to processes (*allow* rules) and those that specify permissions that must be never granted to processes (*never allow* rules).

Types can be grouped into named sets, called *typeattributes*, which may be used inside rules to denote all the types therein. Blocks can also contain *macro* definitions that allow an administrator to abstract a set of rules and to reuse them in different parts of a configuration. Macros can have types as parameters that are instantiated when the macro is called. Moreover, to foster code reuse and modularity, CIL features the construct `blockinherit` that permits a block to *inherit* from another block. Similarly to Object Oriented languages, all the definitions of rules and types in the inherited block are available in the inheriting block.

The most appealing features of CIL with respect to the kernel policy language of SELinux are blocks that enable the administrator build modular configurations, as well as macros and inheritance that allow code reuse. Below, we illustrate CIL's main features through examples.

Consider the following CIL block `house` that declares two types, `man` and `object`, and the permission (the `allow` rule) for processes labeled `man` to read the files labeled `object`:

```
(block house
    (type man)
    (type object)
    (allow man object (file (read))))
```

Intuitively, processes of type `house.man` can read the elements of the class `file` labeled `house.object`. Note that blocks introduce namespaces, and the elements defined therein may be referred to directly within the block itself, or by their qualified name, as done above.

The following block inherits the types `man` and `object` and the relevant permission from the block `house` through the `blockinherit` rule.

```
(block cottage
    (blockinherit house)
    (type garden))
```

Intuitively, blockinherit copies the body of the block house. Thus the qualified names of the copied types become cottage.man and cottage.object. In contrast, the type garden is declared in the block, which is not in house.

Blocks can be nested in other blocks, whose names are visible in the nested (child) block. The outermost block can refer to the elements in the nested ones by qualifying their names. In the block below, the first allow rule refers to the type bird defined in the block nest and to the local type egg. Therefore, the resulting permission is between entities labeled by tree.bird and tree.nest.egg. The result of the last allow rule is identical, where the qualified name tree.bird is used.

```
(block tree
  (block nest
     (type egg)
     (allow bird egg (file (write))))
  (type bird)
  (allow bird nest.egg (file (write))))
```

A global namespace is assumed that includes all the blocks, the global types, and the global permission. For example, in

```
1 (type stranger)
2 (allow stranger public_house.object (file (open)))
3 (block public_house
4     (type object)
5     (allow .stranger object (file (read)))
6     (allow stranger object (file (write))))
```

the name stranger and the fully qualified .stranger in the allow rules both refer to the global type .stranger. Note that if the block public_house declared a type stranger, this declaration would overshadow the global one in the last allow rule, but not the one on line 5 since a fully qualified name is used. Note also that the global allow rule refers to a type declared in the enclosed block.

The administrator can collect a set of rules using a macro-like construct, as shown below.

```
(macro add_dog((type owner)(type companion))
  (type dog)
  (allow owner dog (file (read)))
  (allow companion dog (file (read))))
(block animal_house
  (type man)
  (type cat)
  (call add_dog(man cat)))
```

The macro add_dog adds the type dog to the caller's block and grants a file read permission to the types passed as a parameter (the owner and the dog's companion). Roughly, the content of the macro is substituted for the macro call in the last line where the formal parameters owner and companion are bound to animal_house.man and animal_house.cat.

Name resolution can be rather intricate, especially when constructs are combined in non-trivial ways, such as when inheritance and macros are interweaved. In these cases, configurations may have unexpected behavior (see Section 3 for examples), and lead to misconfigurations that are difficult to spot. This problem is exacerbated by the fact that administrators cannot refer to a formal semantics or specification of some kind, which CIL lacks, but rather to informal documentation that is not fully aligned with the implementation and its recent developments. One contribution of this paper is to provide such a semantics, which is also executable and that serves as a reference implementation. We define it in Appendix A and provide an intuitive account in Section 3.

An administrator can group types into named sets, called type attributes, which may be used in place of a type. The following declares the type attributes pet and not_pet and their types therein.

```
(typeattribute pet)
(typeattributeset pet  (or (animal_house.cat) (animal_house.dog)))
(typeattribute not_pet)
(typeattributeset not_pet (not (pet)))
```

The first type attribute includes the two types animal_house.cat and animal_house.dog. In contrast, the second one includes all the others.

Administrators can also specify which permissions should never be granted to a given type using neverallow rules. The rule below prohibits subjects with type animal_house.cat to read resources of any type not in pet:

```
(neverallow animal_house.cat not_pet (file(read)))
```

The CIL compiler statically checks that no allow rule inside the configuration violates a neverallow rule. In this example the compiler will report an error because animal_house.cat can read the files of type animal_house.man that is in not_pet. Although useful, as we explain below, these checks are insufficient to prevent insecure information flow.

An example from the security domain. Consider the following block mem defined in a CIL configuration designed for OpenWrt powered wireless routers [20].

```
(block mem
  (block read
    (typeattribute subj_typeattr)
    (typeattribute not_subj_typeattr)
    (typeattributeset not_subj_typeattr (not subj_typeattr))
    (neverallow not_subj_typeattr nodedev (chr_file (read)))))
```

This block defines an inner block read and two disjoint type attributes. The first includes the system subjects to be specified later in the code, and the second includes other types. The neverallow rule prevents not_subj_typeattr types from reading a character file of the globally defined type nodedev. The underlying idea is that resources of type nodedev are critical for the system and must be carefully protected. This block shows a typical pattern that administrators use to protect critical resources in CIL using type attributes and neverallow rules.

This pattern offers an extra check. In our example, if we include the following rule

```
(allow untrusted mem.read.nodedev (chr_file (read)))
```

that grants a type untrusted the permission to read a character file of type nodedev, then the CIL compiler raises an error. There are two ways to avoid this error: the administrator may either remove the last rule (because granting the permission is actually dangerous), or add untrusted to subj_typeattr to grant the permission.

However, this pattern is insufficient to control how information flows. For example, consider the following snippet

```
(type untrusted)
(type intermediate_file)
(type deputy)
(typeattributeset mem.read.subj_typeattr deputy)
(allow deputy mem.read.nodedev (chr_file (read)))
(allow deputy intermediate_file (file (write)))
(allow untrusted intermediate_file (file (read)))
```

where the types `untrusted`, `intermediate_file`, and `deputy` are defined, and `deputy` occurs within `mem.read.subj_typeattr`. Now, a leak may occur if a subject in `subj_typeattr` reads a character file of type `nodedev` and forwards information, via `intermediate_file`, to an arbitrary process of type `untrusted`, which is permitted to read that information by the given `allow` rules.

Preventing information flow. Currently, CIL does not prevent transitive information flows between types, like the one above from `nodedev` to `untrusted`. The goal of our work is to extend it with a DSL, dubbed *IFL*, to express information flow control requirements. We call the resulting language IFCIL. In addition, we endow IFCIL with a mechanism for statically checking that a configuration satisfies the stated requirements. Our extensions provide administrators with an extra, automatic check when defining rules that grant or deny information flows from a critical resource.

We provide some intuition on our extension by adding the following lines to the `mem` block above:

```
(typeattribute ind_subj_typeattr)
(typeattribute not_ind_subj_typeattr)
(typeattributeset not_ind_subj_typeattr (not ind_subj_typeattr))
;IFL; ~(nodedev +> not_ind_subj_typeattr) ;IFL;
```

The first three lines introduce two type attributes `ind_subj_typeattr`, and `not_ind_subj_typeattr`, which are declared disjoint. The last line, enclosed between the `;IFL;` markers is IFL *annotation* that specifies the information flow requirement that no information can flow from `nodedev` to `not_ind_subj_typeattr`. This annotation is given as a CIL comment that is used by our verification tool, but is completely ignored by the standard CIL compiler. Thus, an IFCIL configuration is still a CIL configuration.

Note that IFL enables administrators to use a pattern similar to that used with `neverallow`, preventing `not_ind_subj_typeattr` types from getting information from a character file of type `nodedev`. In this way, our tool warns the administrator of the information leakage from `nodedev` character files illustrated above.

3 Formalizing CIL

The official CIL documentation [38] does not formally describe CIL's syntax and semantics. We will first give some paradigmatic examples illustrating why having a semantics is of critical importance. Afterwords, we give our formal semantics, evaluate its adequacy, present the CIL compiler bugs discovered, and describe how their corrections impacted our semantics. However, one bug and several intricacies still remain unaddressed, which, in our opinion, may lead to a misconfiguration and unexpected behavior.

3.1 Extracting Principles from Thorny Examples

We investigate the intricacy of CIL's semantics through some representative examples. The source of many of the complexities stems from name resolution and the order in which the transformations prescribed by CIL's commands are executed. To resolve ambiguities, we experimented with CIL's compiler, and we compared our findings with the official documentation. Moreover, when the documentation was lacking or there was conflicting information, we contacted the CIL developers as the ultimate reference for the intended behavior. We also draw from these experiments some principles that drive our semantics.

We first observe that both static and dynamic binding have a role in the CIL's name resolution strategy.[1] Moreover, some name declarations overshadow others. In particular, a name used inside

[1] Static binding prescribes that the free names (not defined locally in the macro) are resolved with the closest declaration within the block nesting in the code, while dynamic binding prescribes that the free names are resolved within the caller's block [16].

```
(type a)
(block A
  (type a)
  (macro m ()
    (type a)
    (allow a a (file (read)))))
(block B
  (type a)
  (block C
    (call A.m)))
```

(a) Macro call scoping rules.

```
(block A
  (type a)
  (block B
    (allow a a (file (read)))))
(block C
  (type a)
  (block D
    (blockinherit A.B)))
```

(b) Block inheritance scoping rules.

```
(type a)
(macro m((type x))
  (type b)
  (allow x b (file (read))))
(block A
  (call m(a)))
(block B
  (type a)
  (blockinherit A))
```

(c) Mixing macro calls and inheritance.

```
(type a)
(type b)
(macro m()
  (call m1))
(block A
  (macro m1()
    (allow a a (file (read))))
  (block B
    (call m)))
(block C
  (type a)
  (macro m1()
    (allow b b (file (read))))
  (block D
    (blockinherit A.B)))
```

(d) Macro names resolution depends on inheritance.

```
(block A
  (type a)
  (block B
    (macro m()
      (allow a a (file (read))))))
(block C
  (type a)
  (block D
    (blockinherit A.B)))
(block E
  (blockinherit C.D)
  (call m))
```

(e) Closures update upon inheritance.

```
(macro m1 ((type x) (type y))
  (type a)
  (allow x y (file (read))))
(macro m2 ((type x) (type y))
  (type b)
  (allow x y (file (read))))
(macro m3 ((type x) (type y))
  (type c)
  (allow x y (file (read))))
(block A
  (call m1 (B.b, B.c))
  (block B
    (call m2 (b, a))
    (call m3 (c, a))))
```

(f) Parameters resolution.

Fig. 1. Examples of thorny CIL configurations

a macro is resolved by considering different namespaces in a fixed order, as well as parameters and names defined in the macro itself.

Example 3.1. Consider the macro call in block `C` of Figure 1a. When evaluating the allow rule, the name `a` is resolved as the type declared inside the macro. Since the declaration is copied in the block `B.C`, the allow rule predicates on entities of type `B.C.a` because of line 5. Instead, if this line is omitted then `a` is resolved as `A.a` because the definition at line 3 takes precedence (this amounts to a kind of closure of the macro, which explicitly contains all the references to the free names of the enclosing namespace). If both line 5 and 3 are removed, the caller's namespace takes precedence, resulting in `B.a` being allowed to read files of type `B.a`. Finally, if we drop lines 5, 3 and 8, the global environment is consulted, evaluating `a` as `.a`.

We can immediately pinpoint some underlining principles.

Principle 3.2. *Both static and dynamic binding strategies are used to resolve CIL names.*

Principle 3.3. *When resolving macros, names are resolved using contexts in the following order: the names defined inside the macro itself; the closure of the macro; the namespace of the caller; and finally the global environment.*

Names in inherited rules are also resolved using both dynamic and static scoping, but the declarations in the namespace of the inheriting block overshadow the ones where the inherited block is defined.

Example 3.4. Consider the allow rule in Figure 1b. When inheriting the block $A.B$, thereby copying the allow rule inside the block $C.D$, the occurrences of the name a are resolved in the context of the blockinherit rule (namely, in $C.D$). The name is resolved in the block A (where $A.B$ is defined), only if it is not defined in $C.D$. The outcome of this code is that (i) entities labeled as type $A.a$ can read files of type $A.a$ (from the allow rule in block $A.B$); and (ii) entities labeled as type $C.a$ can read files of type $C.a$ (from the same allow rule once copied inside block $C.D$). If the type declaration in line 6 were missing, then the result would have been that entities labeled as type $A.a$ can read files of type $A.a$. As with macros, the global environment is only checked when there are no other declarations.

Principle 3.5. *Dynamic scoping takes precedence over static scoping when evaluating* `blockinherit` *statements.*

Principle 3.6. *When resolving names, the global environment is the default namespace.*

Since macro calls can be inherited, the two behavior described above can mix in subtle ways.

Example 3.7. Consider the configuration in Figure 1c. Here the block `B` inherits from `A`, which calls the macro `m`. There are two plausible orders in which macro calls and inheritances can be resolved, and the choice determines which name the parameter `x` is bound to when the `allow` rule is copied in `B`. If the macro call is resolved before inheritance, then `x` is bound to the global `.a` (since `a` is undefined in `A`). If instead the inheritance is resolved first, then the call instruction is copied inside `B` and `x` is bound to `B.a`. This is CIL's actual behavior, but the reference guide is unclear about this choice.

As a result, inheritance affects the namespace where macro names and parameters are resolved.

Example 3.8. Consider the example in Figure 1d. As in the previous example, we must first resolve inheritance, therefore copying `(call m)` into the block `C.D`. At this point, evaluating the call to `m` requires resolving the name `m1`, which is indeed defined in `C`, leading to elements of (the single) type `b` being capable of reading files labeled with `b`.

Principle 3.9. *Declaration inheritance happens before macro calls evaluation.*

Identifying the closure of an inherited macro definition is another non-trivial aspect of CIL.

Example 3.10. Consider the call to macro `m` in block `E` in Figure 1e. Inheritance is resolved first, therefore copying the definition of `m` from the block `A.B` to `C.D`, and then from `C.D` to `E`. At this point, the macro is called, and the allow instruction is evaluated. For resolving the name `a`, the static binding is considered first, i.e., the macro's closure is consulted. There are two possibilities for determining the macro's closure, both acceptable: either one considers `A.B` (where `m` is originally defined) or `C.D` (where `m` is copied). The CIL compiler resolves `a` as `C.a`, i.e., the definition of `a` that is syntactically closest to the *copy* of the definition of the macro. The final result is that entities of type `C.a` can read files of the same type.

Nevertheless, the original closure of `m` would be used if line `7` were removed, resulting in a permission for the entities of type `A.a`. Actually, the closure of an inherited macro is a stack with the inheriting namespace on top and the original namespace at the bottom.

Principle 3.11. *The closure of a macro is determined by the list of namespaces in which it is defined and copied by* `blockinherit` *statements.*

We already inspected the order of resolution between blockinherit and call. Indeed, also when considering macro calls, the propagation of declarations and the evaluation of allow rules does not happen simultaneously. Types copied from a macro can be passed as parameters to other macros and we noticed that resolving their names might not be done in a strict order, as exemplified below. Yet this is fully supported by the compiler and is as intended by the developers.

Example 3.12. In the code in Figure 1f, every macro call uses the types copied from the other macros, and therefore resolving them is impossible in any chosen strict order. Nevertheless, all the names are available when evaluating the copied allow rules, and the resulting permissions are that `A.B.b` can read both `A.a` and `A.B.c`, which in turn can read `A.a`. We will model this behavior by propagating the declarations in a preprocessing phase before evaluating the allow rules.

Principle 3.13. *Regardless of the ordering, declarations that are copied because of* `blockinherit` *or* `call` *statements are always available as parameters for macros.*

In conclusion, the principles we distilled provide some insight into the decision choices made by the developers. On the one hand, these principles guide us in defining the semantics we present next; on the other hand, they can help system administrators to clarify some ambiguities and corner cases that can occur in their configurations.

3.2 Semantics of CIL

To clarify the effect of CIL configurations on the operating system, and to provide a formal basis for IFCIL and for verification, we provide a formal semantics for CIL. Our semantics focuses on the type enforcement fragment of the language, which is its most used part (see the real-world CIL configurations in Section 6.2), and maps each system type to its set of permissions. Our semantics is implemented in OCaml, available online [9], and it is used in our verification tool IFCILverif.

In CIL, every entity (types, typeattributes, macros and blocks) must be explicitly declared to be used in commands. In the following, we consider separately name declarations and name usages. A CIL declaration associates an entity with a string of alphanumerical characters that we call a *declared name*. Let $DN \ni dn, dn', dn'', \dots$ be the set of declared names. For convenience, in our examples we adopt separate names for each class of entities: we use `a`, `b`, `c`, ... as names for types and typeattributes; `A`, `B`, `C`,... for blocks; `m`, `m1`, `m2` for macros.

When referring to entities in commands, their declared name *dn* may be prefixed by a path of names encoding the nesting of blocks that identifies where *dn* has been declared. For example, `C.D.a` identifies the type name `a` declared within the block `D`, which in turn occurs within `C`. Also, `.A.B` identifies the block `B` declared in `A`, introduced in the global environment, which we explicitly represent by the distinguished symbol #, so the name above will be written `#.A.B`. We call *qualified names* these lists of declared names separated by dots, and we let $QN \ni qn, qn', qn'', \dots$ be the set of qualified names. The qualified names that start with # are called *globally qualified names*. Let $GN \ni gn, gn', gn'', \dots$ be the set of globally qualified names. Finally, we call *namespace*, denoted by $ns, ns', ns'' \dots \in NS$, either # or the globally qualified name of a block. Note that the global environment is the namespace #. We also let $x, x', x'', \dots \in X$ be names used for formal parameters of macros, used as if they where qualified names.

CIL rules, declarations and commands are defined as follows, where $\widetilde{\cdot}$ denotes lists of qualified names, R is a set of rules, and M is a set of rules that contains no block, blockinherit, or macro rules.

$$\begin{aligned}
rule &::= declaration \mid command\\
declaration &::= (\textbf{block}\ dn\ R) \mid (\textbf{typeattribute}\ dn) \mid (\textbf{type}\ dn) \mid (\textbf{macro}\ dn\ (\widetilde{x})\ M)\\
command &::= (\textbf{allow}\ qn\ qn\ (class\ (perms))) \mid (\textbf{typeattributeset}\ qn\ (expr))\\
&\quad\ \mid (\textbf{call}\ qn\ (\widetilde{qn})) \mid (\textbf{blockinherit}\ qn)\\
expr &::= \textbf{all} \mid qn \mid expr\ \textbf{or}\ expr \mid expr\ \textbf{xor}\ expr \mid expr\ \textbf{and}\ expr \mid \textbf{not}\ expr
\end{aligned}$$

A CIL configuration, ranged over by R, R', is itself a set of CIL rules.

Let *CILRules* be the set of all CIL rules. We call *value* $\in$ *Val* of a qualified name the entity it refers to. A value may be either

- a globally qualified name $gn \in GN$ for a type or a typeattribute;
- a pair $(gn, R) \in GN \times 2^{CILRules}$ for a block;
- a tuple $(gn, \tilde{x}, M, \xi, \sigma) \in GN \times \widetilde{X} \times 2^{CILRules} \times 2^{DN} \times \widetilde{NS}$ for a macro.

Intuitively, the value of types (and typeattributes) are their globally qualified names; the value of a block is the pair consisting of its globally qualified name and the CIL rules it contains; finally, the value of a macro is a quintuple: its globally qualified name, the list of its parameters, its rules, the set of its types and typeattributes, and its closure.

The qualified names appearing in commands need to be resolved, i.e., they must be associated with their value, and for that one must find the declaration they refer to. To do this, we define environments mapping a namespace to the declarations that are visible inside that namespace, and we recover the CIL's intricate visibility rules by suitable lists of namespaces called contexts.

We call *frame* $fr : QN \to Val$ a function mapping declared names to the corresponding value (if any). An *environment* is a function

$$\rho \colon NS \to QN \to Val$$

that given a namespace, returns the frame containing the local declarations in the namespace. So, $\rho(ns)(dn)$ returns the value corresponding to dn in the namespace ns.

A namespace refers to the names of an enclosed namespace by suitably qualifying them. To ensure this, we constrain ρ to be such that the following holds:[2]

$$\rho(ns)(qn.dn) = \rho(ns.qn)(dn).$$

[2] In the OCaml implementation of our semantics, we force environments to fulfill this constraint by propagating the updates of ρ accordingly.

For example, if the block A contains a block B that defines the type a, not only the pair $(a, \#.A.B.a)$ is in the frame $\rho(\#.A.B)$, but $(B.a, \#.A.B.a)$ is also in $\rho(\#.A)$. Indeed, an allow rule contained in the block A can use the qualified name $B.a$ for referring to the type in B.

Given a configuration R, its *environment* is built by inspecting the declarations of R (see Figure 2b). We define some useful operations for updating frames and environments. Given a frame fr and a set of declared names ξ, let $fr \setminus \xi$ be the frame where the values for the names in ξ are removed, and $fr \cap \xi$ be the frame only recording the values for the names in ξ. Formally:

$$fr \setminus \xi = \{(dn, v) \in fr \mid dn \notin \xi\} \qquad fr \cap \xi = \{(dn, v) \in fr \mid dn \in \xi\}\,.$$

An environment ρ is updated to $\rho[ns \mapsto fr]$ by substituting the binding for a given namespace ns with a new frame $fr \colon QN \rightarrow Val$. Formally:

$$\rho[ns \mapsto fr](ns') = \begin{cases} fr & \text{if } ns' = ns \\ \rho(ns') & \text{otherwise.} \end{cases}$$

The second auxiliary definition is a lookup function on an environment ρ that searches backwards for the value of a qualified name along the path from a given namespace to #: this corresponds to a static binding strategy.

$$\overline{\rho}(ns.dn)(qn) = \begin{cases} \rho(ns.dn)(qn) & \text{if } qn \in dom(\rho(ns.dn)) \\ \overline{\rho}(ns)(qn) & \text{otherwise, if } ns \neq \#. \end{cases}$$

Notice that this defines a partial function as $\overline{\rho}(\#)$ is always undefined, i.e., the global environment # is not consulted when resolving names. This replicates the CIL compiler's behavior, as the static scoping does not apply to the global environment, which has a special visibility rule.

Our third auxiliary definition is that of *context* σ, which is a list of namespaces, enclosed in square brackets: we write $[]$ to denote the empty list and $ns :: \sigma$ to denote the list with head ns and tail σ. By abusing the notation, we will also apply $\overline{\rho}$ to a context σ (and a qualified name qn) rather than to a namespace ns:

$$\overline{\rho}(ns :: \sigma)(qn) = \begin{cases} \overline{\rho}(ns)(qn) & \text{if } qn \in dom(\rho(ns)) \\ \overline{\rho}(\sigma)(qn) & \text{otherwise} \end{cases} \qquad \text{and} \qquad \overline{\rho}([]) = \rho(\#).$$

Note that this extended usage of $\overline{\rho}$ on contexts σ implements CIL's mixed static and dynamic lookup strategy (as prescribed by Principle 3.2): the namespaces in σ are consulted in the order in which they appear in a dynamic fashion; the prefixes of ns in σ encode static scoping; and the global environment # is only considered when the context σ fails in resolving the given name qn (as prescribed by Principle 3.6).

With all these ingredients, we can now focus on CIL's semantics. A preprocessing step helps to determine the scope of declarations, thereby supporting the resolution of names. Roughly, we normalize the ruleset and the environment ρ by propagating the declarations from the inherited to the inheriting blocks, and from macros to macro callers. This preprocessing makes declarations available when evaluating commands, as prescribed by Principle 3.13. We compute this normal form using a sequence of three transformations $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ that modify the rules of the configuration and the environment. We provide a high-level overview of these transformations, their detailed formalization is given in Appendix A.

$\mathcal{T}_1$ replaces globally qualified names of the blocks for the qualified names in the `blockinherit` rules, and builds an initial environment ρ by inspecting the declarations in the configuration.

$\mathcal{T}_2$ updates the environment ρ by propagating the declarations from inherited to inheriting blocks, and by updating the closure of macros while inheriting them.

```
1  (type a)
2  (type b)
3  (type c)
4  (block A
5    (type c)
6    (macro m1()
7      (allow a b (file (read))))
8    (block B
9      (type b)
10     (macro m()
11       (type c)
12       (call m1))
13     (call m)))
14 (block C
15   (type a)
16   (macro m1()
17     (allow b a (file (read))))
18   (block D
19     (blockinherit A.B)
20     (allow b c (file (read)))))
```

(a) A CIL configuration.

$$\begin{aligned}
\rho(\#) &= \{(\mathsf{a}, \#.\mathsf{a}), (\mathsf{b}, \#.\mathsf{b}), (\mathsf{c}, \#.\mathsf{c}), (\mathsf{A}, R_{\mathsf{A}}), (\mathsf{C}, R_{\mathsf{C}})\}\\
\rho(\#.\mathsf{A}) &= \{(\mathsf{c}, \#.\mathsf{A.c})\\
&\qquad (\mathsf{m1}, \epsilon, \{\\
&\qquad\qquad \texttt{(allow a b (file (read)))},\\
&\qquad\qquad \}, \emptyset, [\#.\mathsf{A}]),\\
&\qquad (\mathsf{B}, R_{\mathsf{B}})\}\\
\rho(\#.\mathsf{A.B}) &= \{(\mathsf{b}, \#.\mathsf{A.b}), (\mathsf{m}, \epsilon, M_{\mathsf{m}}, \{\mathsf{c}\}, [\#.\mathsf{A.B}])\}\\
\rho(\#.\mathsf{C}) &= \{(\mathsf{a}, \#.\mathsf{C.a}),\\
&\qquad (\mathsf{m1}, \epsilon, \{\\
&\qquad\qquad \texttt{(allow b a (file (read)))}\\
&\qquad\qquad \}, \emptyset, [\#.\mathsf{C}]),\\
&\qquad (\mathsf{D}, \{\texttt{(blockinherit \#.A.B)},\\
&\qquad\qquad \texttt{(allow b c (file (read)))}\})\}\\
\rho(\#.\mathsf{C.D}) &= \{\}
\end{aligned}$$

(b) A CIL environment.

Fig. 2. Example of normalization.

$\mathcal{T}_3$ updates ρ by propagating the declarations from macros to their callers' namespaces (the order is required by our Principle 3.9).

Example 3.14. Consider the CIL configuration in Figure 2a. After the first transformation $\mathcal{T}_1$, the name $A.B$ in the blockinherit statement at line 19 is resolved. The line is therefore substituted with (**blockinherit** #.A.B). The environment ρ in Figure 2b is created, where we use R_x and M_y for the rules inside a block x and a macro y when their name is unique.

Then, $\mathcal{T}_2$ propagates the declared names from inherited blocks, thus updating ρ to ρ',

$$\rho' = \rho[\#.\mathsf{C.D} \mapsto \rho(\#.\mathsf{C.D}) \cup \{(\mathsf{b}, \#.\mathsf{C.D.b}), (\mathsf{m}, (\#.\mathsf{C.D.m}, \epsilon, M_{\mathsf{m}}, \{\mathsf{c}\}, [\#.\mathsf{C.D}, \#.\mathsf{A.B}]))\}]\,.$$

Finally, $\mathcal{T}_3$ propagates the declarations from macros to the caller's blocks, obtaining ρ'',

$$\rho'' = \rho'[\#.\mathsf{A.B} \mapsto \rho'(\#.\mathsf{A.B}) \cup \{(\mathsf{c}, \#.\mathsf{A.B.c})\},\ \#.\mathsf{C.D} \mapsto \rho'(\#.\mathsf{C.D}) \cup \{(\mathsf{c}, \#.\mathsf{C.D.c})\}]\,.$$

The semantics $[\![R]\!]$ of a CIL configuration R is a graph $G = (N, ta, A)$. The nodes N model the types and the typeattributes (with global names), and the function $ta\colon N \longrightarrow 2^N$ represents the types contained in a typeattribute (assuming $ta(n) = \{n\}$ when n is a type, which will be always the case in our examples). The arcs $A \subseteq N \times 2^O \times N$ model permissions, where O is the set of SELinux operations; we assume that whenever the typeattribute m operates on m', there are arcs (n, o, n'), for all $n \in ta(m)$ and $n' \in ta(m')$. The meaning of (n, o, n') is that the type n is allowed to perform all operations in o on the resources of type n'.

The components of $G = (N, ta, A)$ are computed by three evaluation functions $\mathcal{E}_N, \mathcal{E}_{ta}, \mathcal{E}_A$, respectively. The nodes N are the globally qualified names in ρ.

$$\mathcal{E}_N(\rho) = \{gn \mid \exists qn.\ gn = \rho(\#)(qn)\}$$

The auxiliary function $cdm(m, \rho, \sigma)$ collects all the declarations copied from the called macro m (and from the macro it calls in turn).

$$cdm(m, \rho, \sigma) = \xi \cup \bigcup_{(\textbf{call } m'(\widetilde{qn})) \in M} cdm(m', \rho, \sigma' \cdot \sigma) \qquad \text{where } (gn, \widetilde{x}, M, \xi, \sigma') = \overline{\rho}(\sigma)(m)$$

Note that cdm collects all the ξ of the chain of called macros, and that macro names are resolved according to the closure if possible, and in the dynamic context otherwise (as σ' is prefixed to σ).

The arcs in A are extracted from the *allow* rules: their source and target are the types in the rule, and the label contains the permissions. Moreover, $\mathcal{E}_A$ evaluates the `call` and `blockinherit` rules to recover the arrows from the allow rules in the called macros and in the inherited blocks.

$$\mathcal{E}_A((\textbf{block } dn\ R), \rho, ns :: \sigma) = \mathcal{E}_A(R, \rho, ns.dn :: \sigma) \tag{1}$$

$$\mathcal{E}_A((\textbf{allow } qn\ qn'\ (class(perms))), \rho, \sigma) = \{(\overline{\rho}(\sigma)(qn), (class, perms), \overline{\rho}(\sigma)(qn'))\} \tag{2}$$

$$\mathcal{E}_A((\textbf{blockinherit } ns.dn), \rho, \sigma) = \mathcal{E}_A(R, \rho, \sigma \cdot [ns]) \quad \text{where } (ns.dn, R) = \overline{\rho}(ns)(dn) \tag{3}$$

$$\begin{aligned} \mathcal{E}_A((\textbf{call } qn\ (\widetilde{qn'})), \rho, ns :: \sigma) &= \mathcal{E}\text{-}Call_A(M[\widetilde{x} \mapsto \widetilde{gn'}], \rho', \sigma' \cdot (ns :: \sigma), ns^f, \xi) \\ &\text{where } (gn, \widetilde{x}, M, \xi, \sigma') = \overline{\rho}(ns :: \sigma)(qn) \text{ and } \widetilde{gn'} = \overline{\rho''}(ns :: \sigma)(\widetilde{qn'}), \\ &\text{with } \rho' = \rho[ns^f \mapsto \rho(ns) \cap cdm(qn, \rho, ns :: \sigma)] \\ &\text{and } \rho'' = \rho[ns \mapsto \rho(ns) \setminus cdm(qn, \rho, \sigma' \cdot (ns :: \sigma))] \end{aligned} \tag{4}$$

$$\mathcal{E}_A(R, \rho, \sigma) = \bigcup_{rule \in R} (\mathcal{E}_A(rule, \rho, \sigma)\} \tag{5}$$

$$\mathcal{E}_A(rule, \rho, \sigma) = \emptyset \qquad \text{if } rule \neq allow, block, call, blockinherit \tag{6}$$

Some comments are in order. The semantics of a block is simply the semantics of its content. An allow rule defines an arrow with source and target determined by the evaluation of the respective types. Inheriting a block requires giving a semantics to the inherited rules, whose names are resolved in the current context σ, if possible, otherwise in the namespace ns of the inherited block (the precedence of dynamic over static scoping of Principle 3.5 is realized by postfixing ns to σ). The semantics of `call` is obtained through the auxiliary function $\mathcal{E}\text{-}Call_A$ that evaluates the content of the macro where the actual parameters $\widetilde{gn'}$ are substituted for the formal ones $\widetilde{x}$. Note that parameters are evaluated in a special environment ρ'' pruned of the names copied by the same macro that we are calling. This solves a bug in previous versions of the compiler that we have reported and is now fixed, see Section 3.4. Note also that when calling $\mathcal{E}\text{-}Call_A$, the closure is prefixed to the actual context, thus enforcing the overshadowing of Principle 3.3. Moreover, a new environment ρ' is obtained by adding a frame for a fresh namespace ns^f that associates the names copied from the macro to their definition in the caller's namespace. This is because names defined inside the macro are resolved locally and not according to the closure (Principle 3.3). Thus they must be bound to the declaration copied into the caller. Finally, the arcs associated with a set of rules are the union of those of the single rules, while rules defining no arrows return the empty set.

The semantics of a rule inside a called macro $\mathcal{E}\text{-}Call_A(r, \rho, \sigma, ns^f, \xi)$ is similar to the one of $\mathcal{E}_A$, but locally defined names that have been copied in the caller's namespace can be accessed via the frame for the fresh namespace ns^f which is consulted first.

$$\begin{aligned} \mathcal{E}\text{-}Call_A((\textbf{call } qn\ (\widetilde{qn'})), \rho, \sigma, ns^f, \xi) &= \mathcal{E}\text{-}Call_A(M[\widetilde{x} \mapsto \widetilde{gn'}], \rho, \sigma' \cdot \sigma, ns^f, \xi') \\ &\text{where } \overline{\rho'}(ns^f :: \sigma)(qn) = (gn, \widetilde{x}, M, \xi', \sigma') \text{ and } \widetilde{gn'} = \overline{\rho'}(ns^f :: \sigma)(\widetilde{qn'}), \\ &\text{with } \rho' = \rho[ns^f \mapsto \rho(ns^f) \cap \xi] \\ \mathcal{E}\text{-}Call_A(rule, \rho, \sigma, ns^f, \xi) &= \mathcal{E}_A(rule, \rho', ns^f :: \sigma) \end{aligned}$$

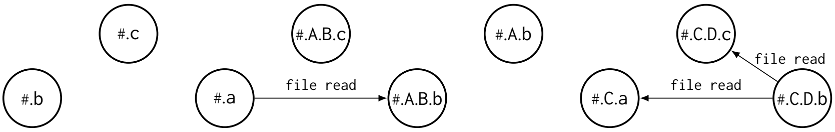

Fig. 3. Semantics of the CIL configuration of Example 3.14.

$$\text{where } \rho' = \rho[ns^f \mapsto \rho(ns^f) \cap \xi] \text{ if } rule \neq call$$

$$\mathcal{E}\text{-}Call_A(M, \rho, \sigma, ns, \xi) = \bigcup_{rule \in M} \mathcal{E}\text{-}Call_A(rule, \rho, \sigma, ns^f, \xi)$$

The definition of $\mathcal{E}_{ta}$ has the same rules (1), (3), (4), and (5) with the obvious changes, and the following two rules where $\overline{\rho}(\sigma)(expr)$ resolve the names occurring in $expr$.

$$\mathcal{E}_{ta}((\textbf{typeattributeset } qn \ (expr)), \rho, \sigma) = \{(\overline{\rho}(\sigma)(qn), \overline{\rho}(\sigma)(expr))\}$$

$$\mathcal{E}_{ta}(rule, \rho, \sigma) = \emptyset \qquad \text{if } rule \neq block, call, blockinherit, typeattributeset \,.$$

Finally, the semantics of a CIL configuration R with environment ρ is

$$[\![R, \rho]\!] = (\mathcal{E}_N(\rho), \mathcal{E}_{ta}(R, \rho, [\#]), \mathcal{E}_{ta}(R, \rho, [\#])).$$

Example 3.15. Figure 3 displays the semantics of the normalized configuration in the Example 3.14, with environment ρ''.

3.3 Adequacy of the Formalization

We have defined CIL's semantics to reflect both the explanations given by the reference manual and the operational semantics defined by the compiler. However, the documentation and the compiler sometimes disagree. In addition, the manual has both underspecified and ambiguous cases. When these mismatches occur and when unexplainable behavior arise, we have asked the CIL developers about the intended behavior [7, 8]. Some cases were recognized as compiler bugs and fixed, whereas the developers have promised to update their documentation in other cases [6]. Based on the developers' fixes, we have updated our previous formalization of the semantics. Unfortunately, some of the fixes they made were ad hoc, which may lead to surprises in certain thorny corner cases.

We identified name resolution as the most involved part of CIL's semantics, especially when name resolution interacts with inheritance or macros. The reference manual lacks a description of how the composition of CIL constructs behaves. In particular, the composition of macro calls and block inheritance behaves differently, depending on the order in which they are resolved. In the beginning of this section we presented several configurations with this kind of problem. Since the manual specifies no evaluation order and ignores this problem, we based the adequacy of this part of the semantics entirely on the compiler and on the developers' feedback.

To understand how to correctly compose the semantics of the different constructs, we performed a comprehensive testing of the CIL compiler with complex configurations whose semantics depends on the order in which constructs are evaluated. We then implemented an executable version of our semantics and validated it with both hand-crafted and randomly generated configurations. Our executable semantics, the test cases, a procedure for automatically generating CIL configurations and one for comparing the semantics with the output of the CIL compiler are available at [9].

```
(block A
  (type a)
  (macro m ()
    (type a)
    (allow a a
      (file (read)))))
(block B
  (call A.m))
```

(a) Bug 1.

```
(macro m((type x))
  (type a)
  (allow x x
    (file (read))))
(block A
  (call m(a)))
```

(b) Bug 2.

```
(type a)
(typeattribute b)
(typeattribute c)
(typeattributeset b (not c))
(typeattributeset c b)
(allow b b (file (read)))
(allow c c (file (read)))
```

(c) Bug 3.

```
(type a)
(type b)
(macro m1((type x))
  (type a)
  (allow x x (file (read))))
(macro m2((type x))
  (type b)
  (allow x x (file (read))))
(block A
  (call m1(b))
  (call m2(a)))
```

(d) A circular case similar to bug 2.

```
(type a)
(macro m((type x))
  (type a)
  (allow x x (file (read))))
(block A
  (call m(A.a))
  (block A
    (call m(a))))
```

(e) Another circular case similar to bug 2.

```
(block A
  (type a)
  (macro m
    (allow a a (file (read)))))
(block B
  (type a)
  (call A.m))
```

(f) Static scoping overtakes dynamic scoping.

```
(type a)
(macro m
  (allow a a (file (read))))
(block B
  (type a)
  (call m))
```

(g) Dynamic scoping overshadows global environment.

Fig. 4. Examples for the reported compiler bugs, and some inconsistencies with the developers' fixes.

3.4 Fixing the CIL Compiler

Our investigation into CIL's semantics revealed various bugs in the compiler: a wrong precedence of visibility of declarations in macros, a counterintuitive usage of macro parameters, and an undefined behavior for ill-defined recursive definition of typeattributes. These bugs were reported to the developers and fixed in more recent versions of the compiler [6]. We also reported to them that the global environment is not handled as a unnamed block containing all the others, which one would naturally expect. However, CIL's developers did not address this issue, which in our opinion may lead to unexpected behavior.

The first bug is due to a mismatch between the compiler's behavior and the reference manual. Consider the configuration in Figure 4a. According to the manual, types defined inside the macro should be checked before those defined in the namespace where the macro is defined. Hence, when copying the `allow` rule from `m` to `B`, we expect the type `a` to be resolved as `B.a`. But it is resolved instead as `A.a`. CIL's developers agreed that this is a compiler bug and fixed it. As a result, now names are overshadowed as described in the manual and formalized in our semantics.

The second bug is instead a counterintuitive behavior, but it does not explicitly contradict the documentation. For example, the configuration in Figure 4b seems impossible to resolve because the type `a` defined inside `m` is passed to `m` itself as a parameter. However, this was not deemed erroneous by the compiler. Namely, the type `a` is copied from the macro `m` to the block `A` and then passed as parameter to `m` itself, i.e., the macro depends on its defined types in a circular way. In a similar puzzling way, if another type named `a` is defined, e.g., in the global environment, it is overshadowed by the type copied from the macro. Also this has been fixed by the developers of the CIL compiler by adding a kind of occur check that forbids passing types copied from a macro `m` as parameters for `m` itself. This is enforced by our semantics by restricting the environment accordingly through the use of ‘\’ over frames. However, the ad hoc solution taken by the developers only concerns cases like the one in Figure 4b. But there are other circular situations, as shown in Figures 4d and 4e, that the CIL developers did not address. In the first example, both macros `m1` and `m2` provide a type that is used as a parameter for the other macro in a circular way. In the second example, the circularity is between two calls to the same macro. In these examples, no call can be resolved first, and both the copied types overshadow the global ones, yet the resulting permissions are computed by propagating the declarations. Indeed, in both configurations, `A.a` can read files of type `A.a`; and the same holds for `A.b` in Figure 4d, and for `A.A.a` in Figure 4e. We model this through our transformations $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$.

The third bug concerns the treatment of ill-formed recursion in typeattribute declarations. Consider the example in Figure 4c, where the typeattribute `b` should contain all the elements that are not in itself: a contradiction. This configuration is meaningless, but the error is not detected by the compiler, which produces a kernel policy with unspecified behavior. According to the compiler, `a` belongs to `b` but not to `c`, which is again contradictory since `c` is defined to be the same as `b`. Note that such a misconfiguration may arise silently in complex code where typeattributes are set using macros in different places in the code. This compiler misbehavior has been fixed by deeming the previous configuration to be incorrect.

Finally, we discovered a counterintuitive behavior concerning the fact the global environment is not coherently treated as an unnamed block containing all the others, when considering the static scoping. As a result, the semantics of the pair of rules defining a type and a macro behaves differently when defined inside a block versus in the global environment, as in lines `2-4` of Figure 4f and `1-3` of Figure 4g. In the first case, the type `a` is resolved as `A.a`, because static scoping is used instead of the dynamic one; while in the second case, the type `a` is resolved as `B.a` according to the fact that dynamic scoping takes precedence over the global environment (Principle 3.3). We reported this behavior to CIL’s developers, but it remains unaddressed.

4 The policy language IFCIL

This section introduces IFL, our DSL for specifying information flow control requirements. We integrate IFL with CIL, obtaining the policy language IFCIL, where IFL annotations are composed with CIL constructs. In addition, we endow IFCIL with a mechanism for statically checking that configurations satisfy their requirements.

4.1 IFL

The constructs of IFL consider SELinux entities, typically types, and the flow of information between them. Using IFL we define both functional requirements, specifying authorized information flows, and security requirements, prohibiting unwanted, possibly dangerous, information flows.

The language. We use IFL to model how information flows from one node of the graph associated with a type by the semantics, to another node, by listing the nodes traversed in the graph, and the

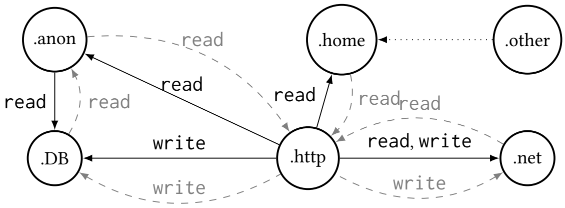

Fig. 5. A simple configuration (black solid arcs) and its information flow diagram (gray dashed arcs); the dotted arc represents inclusion of the target in the typeattribute of the source.

operations allowed on them. This is done by defining a flow *kind P* using the grammar

$$P ::= \texttt{n [o]> n}' \mid \texttt{n +[o]> n}' \mid P_1\ P_2.$$

In this grammar, `n` and `n′` are the starting and the ending nodes in a path. The path is of length 1 for `[o]>`, and of length at least 1 for `+[o]>`. Nodes may also be referenced using the wildcard `*` standing for any node representing a type. The non-empty set $o \subseteq O$ contains a subset of the applicable operations, and o can be omitted when it is the entire set O. The labeled path $P_1\ P_2$ is additionally constrained so that the ending point of P_1 matches the starting one of P_2.

The direction of arrows reflects how information flows in the graph, e.g., `n [write,read]> n′` means that information flows from `n` to `n′` when `n` writes on `n′` or `n′` reads from `n` (the operations in square brackets are the only applicable ones in this step and note too that the curly brackets of sets are omitted). A *one-step information flow* is represented as a single step `n > n′`, whereas a *transitive information flow* is represented by multiple steps `n +> n′`. A kind can also mention intermediate steps, e.g., `n > * > n″ +> n′` specifies that information flows in two steps (through an unspecified node) from `n` to `n″` and then in multiple steps to `n′`.

Kinds are used to constrain the admissible paths of a configuration. Given the semantics $G = (N, ta, A)$ of a configuration, the following construction builds an *information flow diagram*, i.e., a directed graph $I = (N, ta, E)$, where the arcs of E are built as follows. For any arc $(n, o, n') \in A$, E contains: (i) the arc (n, o', n'), where $\emptyset \neq o' \subseteq o$ are the operations of n on n' that cause an explicit information flow from n to n' (e.g., `write`); (ii) the arc (n', o'', n), where $\emptyset \neq o'' \subseteq o$ are the operations of n on n' that cause an explicit information flow from n' to n (e.g., `read`).

The administrator can state the requirements on configurations as assertions over flow kinds given by the grammar

$$\mathcal{R} ::= P \mid {\sim}P \mid P{:}P',$$

The first type of requirement, P, specifies *path existence*, and stipulates the existence of a path of kind P in a given information flow diagram I. The second, ~P, specifies a *path prohibition* and requires that there are no paths in I of kind P. The third is a *path constraint* that requires that every path of kind P in I is also of kind P'.

Figure 5 shows the graph semantics of a simple configuration (with the black solid arcs) and its information flow diagram (with the gray dashed arcs). A dotted arc from a node t to a node t' indicates that t' is in $ta(t)$. We will further discuss this configuration in Figure 6. Intuitively, the entities of type `http` collect information from the network into the database and make data available to the network and to additional entities of type `home`. Information can flow from the network into the database and vice versa, as the configuration satisfies the functional requirements `net +> http +> DB` and `DB +> http +> net` (passing through `anon`). Moreover, the following security requirements are met: `~(DB +> other)` and `DB +> net : DB > anon +> net`. The first states that no information flows from the database to the generic, untrusted types in `other`; the second requirement says that the private information in the database passes through `anon` (where, for example, anonymization takes place) before being delivered in the network.

Semantics. We formalize next when a configuration satisfies a given requirement. We define a path π of an information flow diagram I, and when the path π is of kind P. Intuitively, this holds when the information flow passes through the specified nodes in the correct order as a result of the designated operations.

Definition 4.1 (information flow path and kinds). Let $I = (N, ta, E)$ be an information flow diagram, a *path* in I is the non-empty sequence of arcs in E, i.e., of triples in $N \times 2^O \times N$:

$$\pi = (n_1, o_1, n_2)(n_2, o_2, n_3)...(n_i, o_i, n_{i+1}).$$

We say that π has kind P in I, in symbols $\pi \triangleright_I P$, iff

$$\begin{aligned}
(\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}' \quad &\text{iff}\ \ (m = * \ \vee\ \mathtt{n} \in ta(m)) \wedge (m' = * \ \vee\ \mathtt{n}' \in ta(m')) \wedge o \cap o' \neq \emptyset \\
(\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}' \quad &\text{iff}\ \ (\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}' \\
(\mathtt{n}, o, \mathtt{n}')\ \pi \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}' \quad &\text{iff}\ \ (\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ * \wedge \pi \triangleright_I * \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}' \\
\pi \triangleright_I P_1\ P_2 \quad &\text{iff}\ \ \exists \pi', \pi''.\ \pi = \pi'\pi'' \wedge \pi' \triangleright_I P_1 \wedge \pi'' \triangleright_I P_2
\end{aligned}$$

The definition of $\pi \triangleright_I P$ has four cases. The first case considers a path in the information flow diagram consisting of a single arc of a simple kind; since there exists an operation $\mathtt{op} \in o \cap o'$, the arc `(n, op, n')` can be followed transferring information from `n` to `n'`. The second case reduces $\mathtt{+[}o'\mathtt{]>}$ to $\mathtt{[}o'\mathtt{]>}$. The third case simply iterates the checks along a path longer than one. In the final case, we split a path into two non-empty parts: a prefix satisfying P_1 and a suffix satisfying P_2. Recall that the wildcard $*$ stands for any node and can replace n, n', and n'' above. For example, the first clause can be rewritten as $(\mathtt{n}, o, \mathtt{n}')\ \triangleright_I * \mathtt{[}o'\mathtt{]>}\ \mathtt{n}'$ iff $o \cap o' \neq \emptyset$ because the kind $* \ \mathtt{[}o'\mathtt{]>}\ \mathtt{n}'$ says that information flows from any node to $\mathtt{n}'$.

The predicate $I \models \mathcal{R}$ defined below expresses that a configuration with the information flow diagram I satisfies the requirement $\mathcal{R}$.

Definition 4.2 (validity of a configuration). Let I be an information flow diagram, and let $\mathcal{R}$ be a requirement for a given configuration. We define I to be valid with respect to $\mathcal{R}$, in symbols $I \models \mathcal{R}$, by cases on the syntax of $\mathcal{R}$ as follows:

$$\begin{aligned}
I \models P \quad &\text{iff} \quad \exists \pi \text{ in } I \text{ such that } \pi \triangleright_I P \\
I \models \mathtt{\sim}P \quad &\text{iff} \quad I \not\models P \\
I \models P_1 : P_2 \quad &\text{iff} \quad \forall \pi \text{ in } I \text{ if } \pi \triangleright_I P_1 \text{ then } \pi \triangleright_I P_2
\end{aligned}$$

It is easy to see that the requirements on the configuration in Figure 5 are satisfied.

Expressivity. A path existence constraint expresses a functional requirement, namely that a specific information flow is allowed. If satisfied, this constraint ensures the administrator that the configuration does not prevent the system from performing the desired task. In contrast, a prohibition constraint specifies a security requirement: a configuration obeying it never goes wrong. For example, one can easily specify confidentiality in a Bell-La Padula style, or integrity in the Biba integrity model. Finally, path constraints can express non-transitive properties, like intransitive noninterference. For example $\mathtt{n{+}{>}n'}$: $\mathtt{n{+}{>}n''{+}{>}n'}$ requires that the type `n` cannot transmit any information to $\mathtt{n}'$ unless it is done through $\mathtt{n}''$.

IFL can express (positive and negative) reachability properties with constraints on the paths. Since information flow diagrams are labeled transitions systems, IFL has similarities to temporal logics. Actually, IFL kinds can be expressed as LTL formulas, as the encoding in Section 5.1 shows. However, IFL allows an additional quantification over paths, which can appear only the at top level and is not expressible in LTL.

4.2 IFCIL

Below we introduce the language IFCIL, obtained by integrating IFL into CIL. More precisely, the administrator gives a system specification by directly defining IFL requirements inside blocks and macros. To ensure backward compatibility, requirement definitions are enclosed between `;IFL;` thereby taking the form of CIL comments. IFL requirements are first class citizen of IFCIL. An IFL requirement can use any type or typeattribute name available in the given namespace where the requirement appears, and such names are resolved according to the standard CIL rules. Moreover, CIL's features for structuring configurations are lifted to achieve structured specifications via blocks, inheritance and macros.

The example in Figure 6 is a IFCIL configuration. Lines 1 to 4 contains a macro that defines a new type `anon`, sets a permission, and expresses a requirement. The macro anonymizes the information flowing from entities of the parameter type `x` to `y` by guaranteeing that only `anon` processes can directly access information from `x` (i.e., every information flow from `x` to `y` passes through `anon` as a first step). Four types are defined in lines 6 to 11, and some are collected in the typeattribute `other`. In line 13, the macro is called, thus anonymizing information flows from `DB` to `net`. The allow rules in lines 14-18 set permissions: the http server communicates over the network and interacts with the database (the reading operations are mediated through an anonymizer service). Moreover, the server runs on a system with some home directories that are accessible by the http server. See Figure 5 for a depiction of the resulting permissions and information flows. In addition with the IFL requirement resulting from calling `anonymize`, some other requirements are directly defined in the global environments in lines 20-22. The first two are functional requirements specifying that the http server mediates the communication from the database to the network and vice-versa. Finally, the last rule states that information should not be allowed to flow from the database to entities of types defined outside the current scope.

The syntax of IFCIL is obtained by adding the following rule for IFL requirement to the CIL grammar of Section 3.

$$\textit{command} ::= \texttt{;IFL;}\ \mathcal{R}\ \texttt{;IFL;}$$

We are now ready to define the semantics of IFCIL. The normalization of Section 3 is trivially updated by ignoring the IFL requirement declarations. The semantics of an IFCIL configuration R is a pair $(G, \mathbb{R})$, with $G = [\![R]\!]$ defined as usual, and $\mathbb{R} = \mathcal{E}_{\mathbb{R}}(R, \rho_R, [\#])$, where $\mathcal{E}_{\mathbb{R}}$ is the semantics function with the same rules (1), (3), (4) and (5) of $\mathcal{E}_{\mathbb{A}}$ with the obvious changes, and the following rule, where $\overline{\rho}(\sigma)(\mathcal{R})$ returns the IFL requirement obtained by resolving all the type and typeattribute names appearing in $\mathcal{R}$.

$$\mathcal{E}_{\mathbb{R}}(\texttt{;IFL;}\ \ \mathcal{R}\ \ \texttt{;IFL;}, \rho, \sigma) = \{\overline{\rho}(\sigma)(\mathcal{R})\}$$

Not all configurations satisfy their requirements, and we define below when they do, i.e., when the information flow respects the constraints expressed by the IFL annotations.

Definition 4.3 (correct IFCIL configuration). Let Σ be a (normalized) IFCIL configuration, let $(G, \mathbb{R})$ be its semantics, and let I be the information flow diagram of G. The configuration Σ is correct, in symbols $I \models \mathbb{R}$, iff $I \models \mathcal{R}$ for all $\mathcal{R} \in \mathbb{R}$.

```
1  (macro anonymize((type x) (type y))
2    (type anon)
3    (allow anon x (file (read)))
4    ;IFL; x +> y : x > anon +> y ;IFL;)
5
6  (type DB)
7  (type http)
8  (type home)
9  (type net)
10 (typeattribute other)
11 (typeattributeset other (not (or DB (or http (or anon net)))))
12
13 (call anonymize(DB net))
14
15 (allow http anon (file (read)))
16 (allow http DB (file (write)))
17 (allow http other (file (read)))
18 (allow http net (file (read write)))
19
20 ;IFL; net +> http +> DB ;IFL;
21 ;IFL; DB +> http +> net ;IFL;
22 ;IFL; ~ DB +> other ;IFL;
```

Fig. 6. Example of CIL configuration with IFL annotations.

4.3 IFCIL Extensions

Our IFCIL implementation in Section 6 has several extensions that improve the usability of IFL statements. Some extensions are purely syntactic, e.g., permitting one to compose IFL arrows directly without intermediate nodes. In this way one can write `n +>> n'` for a path from `n` to `n'` of length at least two, or `n >+[read]>> n'` for a path from `n` to `n'` of length at least three and where all the internal steps are due to read operations.

More relevant is our second extension that enables administrators to assign IFL requirements labels that can be used to refer to them later in the specification. The administrator can, for example, replace line 4 of Figure 6 with the following one:

```
;IFL; (S1) x +> y : x > anon +> y ;IFL;)
```

These labels allow administrators to refer to requirements and update them, supporting structured specifications in IFCIL and specification reuse. For example, consider line 13 of Figure 6 where the macro `anonymize` is called that requires the flow to pass through `anon`. By adding the following requirement at the end of the configuration, we can further impose that the information flow from `DB` to `anon` is a `read` operation:

```
;IFL; (S2) DB +> net : DB [read]> anon +> net ;IFL;
```

This makes the requirement `(S1)` just introduced unnecessary as it is subsumed by `(S2)`.

Instead of introducing `(S2)` as a new requirement, the administrator can redefine `(S1)` as follows:

```
;IFL; (S1) DB +> net : DB [read]> anon +> net ;IFL;
```

Note that the same label is used, and the new requirement is intended to replace the old one. However, this is only permitted when the new requirement subsumes the old one, in which case we call it a *refinement*; otherwise our tool raises an error.

$$\text{(node)}\,\frac{-}{n \preceq_n *} \qquad \text{(operation)}\,\frac{o \subseteq o'}{o \preceq_o o'} \qquad \text{(arrow)}\,\frac{-}{\texttt{>} \preceq_a \texttt{+>}} \qquad \text{(o-arrow)}\,\frac{o \preceq_o o' \quad w \preceq_a w'}{(w,o) \ \preceq_{oa} \ (w',o')}$$

$$\text{(P-1)}\,\frac{n_1 \preceq_n n_1' \quad n_2 \preceq_n n_2' \quad (w,o) \preceq_{oa} (w,o)'}{n_1 \ (w,o)\ n_2 \ \preceq_P \ n_1' \ (w,o)' \ n_2'} \qquad \text{(P-2)}\,\frac{n_1 \preceq_n n_1' \quad n_2 \preceq_n n_2' \quad o_1 \preceq_o o' \quad o_2 \preceq_o o'}{n_1 \ \texttt{+[}o_1\texttt{]>} \ * \ \texttt{+[}o_2\texttt{]>}\, n_2 \ \preceq_P \ n_1' \ \texttt{+[}o'\texttt{]>}\, n_2'}$$

$$\text{(P-3)}\,\frac{n_1 \preceq_n n_1' \quad n_2 \preceq_n n_2' \quad o_1 \preceq_o o_1' \quad o_1 \preceq_o o_2' \quad o_2 \preceq_o o_2'}{n_1 \ \texttt{+[}o_1\texttt{]>} \ * \ \texttt{[}o_2\texttt{]>}\, n_2 \ \preceq_P \ n_1' \ \texttt{[}o_1'\texttt{]>} \ * \ \texttt{+[}o_2'\texttt{]>}\, n_2'}$$

$$\text{(P-4)}\,\frac{n_1 \preceq_n n_1' \quad n_2 \preceq_n n_2' \quad o_1 \preceq_o o_1' \quad o_2 \preceq_o o_2' \quad o_2 \preceq_o o_1'}{n_1 \ \texttt{[}o_1\texttt{]>} \ * \ \texttt{+[}o_2\texttt{]>}n_2 \ \preceq_P \ n_1' \ \texttt{+[}o_1'\texttt{]>} \ * \ \texttt{[}o_2'\texttt{]>}\, n_2'} \qquad \text{(comp)}\,\frac{P_1 \preceq_P P_1' \quad P_2 \preceq_P P_2'}{P_1P_2 \ \preceq_P \ P_1'P_2'}$$

$$(\mathcal{R}\text{-1})\,\frac{P \preceq_P P'}{P \preceq P} \qquad (\mathcal{R}\text{-2})\,\frac{P \preceq_P P'}{\sim\! P' \preceq \sim\! P} \qquad (\mathcal{R}\text{-3})\,\frac{P_1 \preceq_P P_1' \quad P_2 \preceq_P P_2'}{P_1' : P_2 \preceq P_1 : P_2'}$$

Fig. 7. Definition of IFL requirement refinement and auxiliary relations.

We formalize the notion of refinement as a preorder over requirements: $\mathcal{R}'$ *refines* $\mathcal{R}$ when $\mathcal{R}' \preceq \mathcal{R}$. We define $\preceq$ in Figure 7 and we use some auxiliary relations. For all of them we implicitly assume reflexivity and transitivity; we write w for unlabeled arrows > or +> and we write (w, o) for the arrows with label o, e.g., +[*read*]> is represented as $(\texttt{+>}, \{read\})$.

First, refinement relations are defined for nodes, operations, and arrows in isolation. The rule (o-arrow) says that composing refined operations and arrows results in a refined labeled arrow. Similarly, the rule (P-1) permits refining a path composed by two nodes and a labeled arrow by refining both the nodes and the labeled arrow. The rule (P-2) states that one can refine a path by specifying the intermediate operations of the path. The rules (P-3) and (P-4) say that a path composed by a subpath of length 1 followed by a possibly longer one (e.g., `n [o]> n' +[o']> n`”) refines a path in which the single step is at the end (e.g., `n +[o]> n' [o']> n`”), and vice versa. The composition of refined paths is a refined path itself, by rule (comp). Finally, we define requirement refinement by stating that: a path existence requirement is refined by refining the path itself (rule ($\mathcal{R}$-1)); a path prohibition is refined by prohibiting a new path that is refined by the original one (rule ($\mathcal{R}$-2)); and a path constraint refinement is defined as the composition of the two above (rule ($\mathcal{R}$-3)).

The following theorem states that refinement is correct, namely that a refined requirement is satisfied by a subset of the information flow diagrams that satisfy the original requirement.

Theorem 4.4. *Let I be an information flow diagram, and let $\mathcal{R}'$ and $\mathcal{R}$ be two IFL requirements such that $\mathcal{R}' \preceq \mathcal{R}$. Then $I \models \mathcal{R}'$ implies $I \models \mathcal{R}$.*

5 Requirement verification

We describe next how we automatically check that a IFCIL configuration respects the given information flow requirements. We rely on model checking, so as to reuse existing verification tools. For this, we first encode a configuration as a Kripke transition system [31] and an IFL requirement as an LTL formula.

5.1 Encoding in temporal logic

A Kripke transition system (KTS) over a set AP of atomic propositions is $K = (S, Act, \longrightarrow, L)$, where S is a set of states, Act is a set of actions, $\longrightarrow \subseteq S \times Act \times S$ is a transition relation, and $L\colon S \longrightarrow 2^{AP}$ is a labeling function mapping nodes to a set of proposition that hold at that node. Paths of K are defined as alternating sequences of states and actions starting and ending with a state.

We associate an IFCL configuration Σ with a KTS with the nodes of Σ as states and the edges of the information flow diagram of Σ as transitions (for technical reasons, transitions are labeled with a single operation), and the type and typeattribute names of Σ as atomic propositions.

Definition 5.1 (Encoding of configurations). Let $I = (N, ta, E)$ be the information flow diagram of a configuration Σ. The corresponding KTS is $K = (N, O, E', \Lambda)$, where

- O is the set of SELinux operations
- $E' = \{(n, op, n') \mid (n, o, n') \in E \wedge op \in o\}$
- $M \in \Lambda(n)$ if $n \in ta(M)$, i.e., n is in the typeattribute M

We encode IFL kinds in a suitable version of LTL [31], where the syntax of formulas ϕ is

$$\phi ::= p \mid (op) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid X(\phi) \mid \phi_1 U \phi_2.$$

We write $w \models_l \phi$ if the path w of K satisfies the LTL formula ϕ; the formal definition is standard and can be found in [31]. Intuitively, w satisfies the atomic proposition p if it starts with a node labeled with p; w satisfies (op) if its first action is $op \in O$; conjunction, disjunction, and negation are as usual; $X(\phi)$ is satisfied by w if its subpath starting from the second state satisfies ϕ; and w satisfies $\phi_1 U \phi_2$ if there exists a node s in w such that the subpath starting from it satisfies ϕ_2 and every subpath starting from a state before s satisfies ϕ_2.

For convenience, in the following we simplify our grammar for the flow kind $P_1 P_2$ and rewrite the grammar from subsection 4.1 in the following equivalent form (recall that the starting node of P in the last two cases is n').

$$P ::= \texttt{n [o]> n'} \mid \texttt{n +[o]> n'} \mid (\texttt{n [o]> n'})P \mid (\texttt{n +[o]> n'})P$$

Definition 5.2 (Encoding of flow kinds). The encoding of flow kinds is defined as follows.

$$\llparenthesis \texttt{n [o]> n'} \rrparenthesis = \mathsf{n} \wedge \bigvee_{\mathsf{op} \in \mathsf{o}} (\mathsf{op}) \wedge X(\mathsf{n}' \wedge \neg X(true))$$

$$\llparenthesis \texttt{n +[o]> n'} \rrparenthesis = \mathsf{n} \wedge \bigvee_{\mathsf{op} \in \mathsf{o}} (\mathsf{op}) \wedge X(\bigvee_{\mathsf{op} \in \mathsf{o}} (\mathsf{op})\ U\ (\mathsf{n}' \wedge \neg X(true)))$$

$$\llparenthesis (\texttt{n [o]> n'})P \rrparenthesis = \mathsf{n} \wedge \bigvee_{\mathsf{op} \in \mathsf{o}} (\mathsf{op}) \wedge X(\llparenthesis P \rrparenthesis)$$

$$\llparenthesis (\texttt{n +[o]> n'})P \rrparenthesis = \mathsf{n} \wedge \bigvee_{\mathsf{op} \in \mathsf{o}} (\mathsf{op}) \wedge X(\bigvee_{\mathsf{op} \in \mathsf{o}} (\mathsf{op})\ U\ \llparenthesis P \rrparenthesis)$$

Note that we use $\neg X(true)$ in our encoding, i.e., the path is complete as there is no next step. This reflects that IFL's semantics is defined on finite paths.

LTL's semantics can be lifted to a KTS K in quite different manners. The best for modeling IFL is as follows: $K \models_l \phi$ iff $\forall w \in W. w \models_l \phi$, where W is the set of all (finite and infinite) paths in K. This paves the way for defining when a KTS satisfies a set of IFL requirements.

Definition 5.3 (Satisfaction of configurations). Let $\mathcal{R}$ be a requirement of a given configuration, let K be a KTS, and let W be the set of paths in K. We define the satisfaction relation $\vdash$ on the syntax of $\mathcal{R}$ as follows:

$$K \vdash P \text{ iff } K \not\models_l \neg\llparenthesis P \rrparenthesis$$

$$K \vdash {\sim}P \text{ iff } K \models_l \neg ⟪P⟫$$
$$K \vdash P : P' \text{ iff } K \models_l \neg ⟪P⟫ \vee ⟪P'⟫$$

We homomorphically extend $\vdash$ to sets of requirements.

Note that the three clauses above mimic the analogous clauses in the Definition 4.2. The first clause says that at least one path satisfies $⟪P⟫$. The second clause says that no path satisfies $⟪P⟫$. The third clause simply formalizes the boolean definition of classical implication.

This correspondence supports the correctness of our verification technique: the notions of validity and satisfaction of configurations coincide.

Theorem 5.4. *Let Σ be an IFCIL configuration with requirements $\mathbb{R}$, let I be its information flow diagram, and let K be the KTS of Σ. Then $K \vdash \mathbb{R}$ if and only if $I \models \mathbb{R}$.*

5.2 Model checking IFCIL

Theorem 5.4 enables us to reuse model checking techniques and tools to automatically verify that a configuration is correct with respect to its information flow requirements. In particular, an LTL model checker provides us with a decision algorithm for $K \models_l \phi$.

In this work, we resort to the classical model checker NuSMV that targets only infinite paths, as usual, whereas Definition 5.3 also considers finite paths. Therefore, we extend K to a KTS K_ι with a distinguished sink state ι and with additional transitions (labeled with every $op \in O$) from every state to ι. Roughly, the satisfaction relation is updated by substituting $X(\iota)$ for $\neg X(true)$. Theorem 5.4 still holds (see Corollary C.6 in Appendix C). This extension enables us to verify the correctness of IFCIL configurations with NuSMV.

The worst case complexity of LTL model checking is unfortunately $2^{O(|\phi|)}O(|S| + |E'|)$ [31], where $|S|$ and $|E'|$ are the number of nodes and number of arcs in the KTS, respectively. In practice we expect the size of the configuration and the number of requirements to grow as the system grows. In contrast, we do not expect the size of each IFL requirement to depend on the system size.

We now specialize the formula above to our encoding. It is easy to see that $|S|$ is equal to the number of type declarations in the configuration, and that $|E'|$ is bounded from above by $|S|\times|S|\times|O|$, where O is the set of SELinux operations. Note that the size of an LTL formula resulting from the encoding of an IFL requirement is linear in the number of names of the requirement. Thus, the complexity of verifying an IFCIL configuration is $\sum_{\phi\in\mathbb{R}} 2^{O(|\phi|)}O(|S|)^2 \times O(|O|)$. Since $|\phi|$ is usually small and does not increase with the configuration size, the complexity grows linearly with the number of requirements and operations, and quadratically with the number of types.

Experiments with our prototype implementation on real-world configurations show that results are obtained in an acceptable amount of time, on the order of seconds, see below.

6 The tool IFCILverif

We now describe our tool IFCILverif that given a IFCIL configuration verifies its correctness with respect to its information flow requirements. Although our tool is currently an optimized prototype, we were nevertheless able to successfully apply it to large, complex, real-world policies.

6.1 Translation to NuSMV

Our tool has a front end that reads a configuration, normalizes it, and then computes its semantics, the associated KTS, and the LTL representation of the requirements, expressed in the NuSMV input language. The result is supplied to the model checker NuSMV, which checks each requirement. Finally the administrator is notified which requirements are satisfied and which are not.

In more detail, IFCILverif takes as input an IFCIL configuration and an associated file where every operation comes with the direction of the information flow it causes. This file is used to build the information flow diagram.

The tool explicitly handles CIL's constructs for defining classes and permissions, and reduces the input configuration to one that only uses the fragment of CIL presented in Section 3. Since other constructs, e.g., those for roles, do not affect requirement satisfiability, the tool just ignores them.

For example, the IFCIL semantics of configuration in Figure 6 is the pair $(G, \mathbb{R})$, where G is the CIL semantics in Figure 5 and $\mathbb{R}$ contains the following requirements:

```
.net +> .http +> .DB,          .DB +> .http +> .net,
~ .DB +> .other,               .DB +> .net: .DB [read]> .anon +> .net
```

It is trivial to derive the KTS K associated with G. To verify the satisfaction of the requirements, we check $K \vdash \mathcal{R}$ for all $\mathcal{R} \in \mathbb{R}$. We only show the case for the last requirement above, i.e., $K \models_l \neg\llparenthesis \texttt{.DB +> .net} \rrparenthesis \vee \llparenthesis \texttt{.DB [read]> .anon +> .net} \rrparenthesis$ where:

$$\llparenthesis \texttt{.DB +> .net} \rrparenthesis = \texttt{.DB} \wedge \bigvee_{\texttt{op} \in O} (\texttt{op}) \;\wedge\; X(\bigvee_{\texttt{op} \in O} (\texttt{op})\; U\; (\texttt{.net} \wedge \neg X(true)))$$

$$\llparenthesis \texttt{.DB [read]> .anon +> .net} \rrparenthesis = \texttt{.DB} \wedge (\texttt{read}) \;\wedge\; X(\texttt{.anon} \wedge \bigvee_{\texttt{op} \in O} (\texttt{op}) \wedge$$
$$X(\bigvee_{\texttt{op} \in O} (\texttt{op})\; U\; (\texttt{.net} \wedge \neg X(true))))$$

The resulting input file for NuSMV represents the nodes of the KTS by variable assignments and transitions as updates of such assignments (using the `next` operator). We briefly comment on the encoding to generate the input file:

- The state variable `t` has the enumeration type that lists all the types in the configuration, plus `sink` (i.e., ι).
- Typeattributes are encoded as symbols and defined as predicates on types.
- The input variable `op` has the enumeration type that lists all the operations in O.
- The transitions are defined in `TRANS`: from each starting node there is an arc to the possible types and typeattributes with the appropriate operation.
- Requirements are expressed in the syntax of NuSMV as defined by $\llparenthesis\_\rrparenthesis$.

IFCILverif then parses the response of NuSMV and answers positively: all the requirements are verified within few seconds.

6.2 Validation

CIL policies. We experimentally assessed our tool on ten real-world CIL policies. The first policy [20] is used in the OpenWrt project, a version of the Linux operating system targeting embedded devices, like network appliances [34]. The second and the third are SELinux example policies, namely *cilbase* [21] and *dspp5* [22], which serve as templates for creating personalized configurations. The remaining configurations are SEAndroid policies: five versions of the Android Open Source Project (AOSP), and two Original Equipment Manufacturer versions for Samsung devices. The SEAndroid configurations have been taken from the ones extracted from disk images by [25] of different builds, and compiled into CIL; the original versions are available at [15]. The analyzed policies have more than ten thousands lines of code, and the first three make extensive use of all CIL's advanced features, in particular macros and blocks.

```
MODULE main

DEFINE
 other := (!((t=DB | (t=http | (t=anon | t=net)))) & !(t=sink);
VAR
 t : { sink , DB , anon , home , http , net };
IVAR
 op : { read , write };
TRANS
 (t=DB -> ((op=read & next(t=anon)) | next(t=sink))) &
 (t=anon -> ((op=read & next(t=http)) | next(t=sink))) &
 (t=home -> (next(t=sink))) &
 (t=http -> ((op=write & next(t=DB)) | (op=write & next(t=net)) | next(t=sink))) &
 (t=net -> ((op=read & next(t=http)) | next(t=sink))) &
 (t=sink -> next(t=sink))

LTLSPEC !((t=DB & X(F t=net)) | (t=DB & op=read & X(t=anon & X(F t=net))))
LTLSPEC !(t=net & X(F t=http))
LTLSPEC !(t=http & X(F t=net))
LTLSPEC !(t=DB & X(F(t=http & X(F t=net))))
LTLSPEC !(t=net & X(F(t=http & X(F t=DB))))
LTLSPEC !(t=DB & X(F other))
```

Fig. 8. NuSMV configuration.

IFL Properties. To illustrate IFL's expressivity, we formalize various properties in IFL that are often considered in the literature, as well as additional domain-specific policies that we designed. Expressing these properties in IFL is easy and the resulting requirements are short, direct, and natural. Afterwards we assess the scalability of IFCILverif on our real-world CIL policies and show that it scales well to large configurations, checking their requirements (given by different subsets of our properties) in a few minutes in the worst cases.

As our first example, consider the following property inspired by Jaeger et al. [28]. They investigated in their work the *trusted computing base* (TCB) of an SELinux configuration and checked from which types information flows to the TCB, identifying those flows that do not compromise security. Using IFCIL, the administrator can straightforwardly restrict the information flows to the TCB to the permitted ones by defining the typeattributes `TCB` and `Harmless`, and by requiring `+> TCB : Harmless +> TCB`.

The second property we consider, *Clark-Wilson integrity* [13], forbids flows from *untrusted* to *critical* nodes, unless a *trusted* node sanitizes the flow. To encode this property in IFL it suffices to enrich configurations with three typeattributes, with the obvious meaning, and to use the following requirement: `untrusted +> critical : untrusted +> sanitizer +> critical`.

The third property is sometimes called an *assured pipeline* [49]. It states that the flow from `a` to `z` must pass through a list of intermediate entities `b,c` ... [24]. Here, it suffices to define requirements of the form `a +> z : * +> b +> c +> ... +> *`.

Our fourth property expresses express the *wrapping of untrustworthy programs* of [49]. It defines requirements stating that all the information flows from (or to) a given type `untrustworthy` must pass through a `verifier` type as a first step, i.e., `untrustworthy > * : * > verifier`.

Our fifth property is *compartmentalization.* To formalize this, assume that nodes are partitioned in groups with specific interfaces. Compartmentalization then states that information flows between groups only through the prescribed interfaces. For example, we can group the nodes of our running example of Figure 5 into `other` and in the additional typeattribute `web`, containing `http`, `net`, `anon`, and `DB`. The interface between them is in yet another typeattribute `other2web`, containing `home`. Then, the intended flows are all the ones satisfying `other +> web : other +> other2web +> web`.

Finally, we propose a new property, *augment-only*, stating that the operations performed by elements of type `a` can only append additional information to targets with type `b`. This property is expressed as `a > b : a [append]> b` and `a +>> b : a +> [append]> b`.

Experimental results. The results of our analyses on the real-world configurations are summarized in Table 1. For each row, the table reports the kind of property considered, the number of requirements of that kind, the time required to compile them, i.e., to compute the semantics and to generate the NuSMV module, and the total time taken to verify them. The tool took about one minute to check the entire OpenWRT configuration (more than 45K lines of code), about three seconds for *cilbase* (about 12k lines), and about seven seconds for *dspp5* (about 15K lines). The performance on SEAndroid policies compiled in CIL are similar, ranging from roughly 15 seconds to less than three minutes. Our analysis reports that some requirements are violated. Among these, the checks on the TCB property show that information flows exist from types that are likely untrusted to types related to the OS security mechanisms, e.g., in *dspp5*, information can flow from `.lostfound.file` to `.sys.fs`. We are investigating whether these types are indeed untrusted and the actual impact that the detected violations have on security. This however requires checking with the developers or reverse engineering to better understand the intended security goals of the analyzed policies.

7 Related Work

Checking the security of the information flow is a problem that has been widely studied since the seminal work by Denning [14], and Goguen and Meseguer [17]. Below, we discuss proposals that target information flow in SELinux policies, also with the help of domain specific languages. We also consider research that addresses verifying information flow in access control languages, and that augment existing programming languages with information flow policies. For a broad survey on the topic, we refer the reader to [43].

Information flow in SELinux. Numerous tools for SELinux policy analysis have been proposed. Many of them are based on information flow, but none targets CIL or explicitly handles the advanced features we consider. These tools can be divided in two categories. The first focuses on predefined tests, searching for specific kinds of misconfigurations. The second supports administrators in querying information flow properties of given policies. Since our tool enables administrators to perform custom analysis, it differs from the proposals in the first category that we briefly survey.

Reshetova et. al. [42] propose SELint, a tool for detecting common kinds of misconfigurations in given SELinux configurations, e.g., the overuse of default types, and the association of specific untrusted types with critical permissions. In contrast to our work, their approach is also specialized for mobile devices.

Radika et. al. [41] analyse SELinux configurations to spot potentially dangerous information flows. They consider an information flow from an entity a to an entity b to be potentially dangerous if a `neverallow` rule prohibits a direct read access from b to a. They propose two tools: the first statically investigates such information flows in configurations, and has been applied to the SELinux reference policy and to the Android policy [18]; the second is a run-time monitor that dynamically tracks information flows in an SELinux system. Our tool does the same kinds of analysis, and also expresses more specific requirements. We can, for example, check for the presence of one-step information flows caused by operations different from those in `neverallow` and of intransitive information flows that pass through a specific path.

Table 1. Performance analysis on tree real-world configurations: *compilation time* is the time for computing the semantics and translating the information flow and the requirements to NuSMV; *verification time* is the time of running the model checker; *total time* is the sum of both phases.

Configuration	Property	Requirements	Compilation Time	Verification Time	Total Time
openWRT 45702 lines 590 types	TCB	1	41,096 sec	5,011 sec	46,107 sec
	assured pipeline	3	40,492 sec	10,227 sec	50,719 sec
	wrap untrustworthy	10	41,171 sec	15,147 sec	56,318 sec
	augment only	2	40,726 sec	9,961 sec	50,687 sec
	total	16	40,754 sec	32,465 sec	73,219 sec
cilbase 11989 lines 293 types	TCB	1	3,157 sec	0,099 sec	3,256 sec
	assured pipeline	4	3,179 sec	0,194 sec	3,373 sec
	wrap untrustworthy	6	3,136 sec	0,227 sec	3,363 sec
	augment only	2	3,158 sec	0,191 sec	3,349 sec
	total	13	3,447 sec	0,515 sec	3,962 sec
dspp5 14782 lines 149 types	TCB	1	5,775 sec	0,546 sec	6,321 sec
	assured pipeline	4	5,515 sec	0,806 sec	6,321 sec
	wrap untrustworthy	8	5,738 sec	1,299 sec	7,037 sec
	total	13	5,766 sec	1,969 sec	7,735 sec
aosp-marlin 10884 lines 769 types	TCB	1	2,619 sec	6,918 sec	9,537 sec
	compartmentalization	1	2,573 sec	2,520 sec	5,093 sec
	CW-lite	1	2,299 sec	7,202 sec	9,501 sec
	total	3	3,488 sec	12,335 sec	15,823 sec
aosp-sailfish-nj 10896 lines 769 types	TCB	1	2,03 sec	6,075 sec	8,105 sec
	compartmentalization	1	2,029 sec	2,515 sec	4,544 sec
	CW-lite	1	2,314 sec	7,174 sec	9,488 sec
	total	3	2,997 sec	12,272 sec	15,269 sec
aosp-sailfish-op 19420 lines 1127 types	TCB	1	6,27 sec	15,699 sec	21,969 sec
	compartmentalization	1	5,892 sec	7,093 sec	12,985 sec
	CW-lite	1	6,149 sec	16,495 sec	22,644 sec
	total	3	6,533 sec	29,756 sec	36,289 sec
aosp-sailfish-pp 25683 lines 1336 types	TCB	1	9,939 sec	24,237 sec	34,176 sec
	compartmentalization	1	9,809 sec	9,908 sec	19,717 sec
	CW-lite	1	10,044 sec	24,131 sec	34,175 sec
	total	3	9,619 sec	45,100 sec	54,719 sec
aosp-sailfish-pq 15695 lines 1337 types	TCB	1	9,72 sec	24,401 sec	34,121 sec
	compartmentalization	1	9,843 sec	9,774 sec	19,617 sec
	CW-lite	1	9,29 sec	24,532 sec	33,822 sec
	total	3	10,265 sec	44,883 sec	55,148 sec
samsung-G 23400 lines 2033 types	TCB	1	11,452 sec	46,403 sec	57,855 sec
	compartmentalization	1	11,693 sec	12,868 sec	24,561 sec
	CW-lite	1	11,754 sec	45,493 sec	57,247 sec
	total	3	11,424 sec	87,606 sec	99,030 sec
samsung-P 24909 lines 2127 types	TCB	1	10,365 sec	165,376 sec	175,741 sec
	compartmentalization	1	10,399 sec	165,451 sec	175,850 sec
	CW-lite	1	11,012 sec	164,882 sec	175,894 sec
	total	3	10,427 sec	165,634 sec	176,061 sec

Jaeger et. al. [28] analyze the SELinux example policy for Linux 2.4.19, focusing on integrity properties. They determine which entities are in the TCB and analyze their integrity by focusing on transitive information flow. As discussed above, we let the administrator specify the TCB and the desired requirements while developing the configuration, rather than deriving the TCB after the policy is implemented.

We now briefly discuss the proposals in the second category that are closest to ours. These proposals neither directly work on structured CIL configurations nor they offer real support for advanced features of this language. Moreover, they do not allow labeling configurations with requirements that interact with the language constructs. All the properties they consider are global. In contrast, our proposal works directly on structured CIL configurations and our requirements are first class citizens in IFCIL.

Guttman et al. [24] propose a formal model of SELinux access control, based on transition systems, and provide an LTL model checking procedure to verify that a configuration satisfies the security goals specified by the administrator. The security goals they consider are non-transitive information flow properties: they verify that every information flow between two given SELinux entities (e.g., users, types, roles) passes through a third entity. As discussed above, IFCIL expresses these requirements, also with conditions about the operations occurring in the information flow. In contrast, we do not consider exceptions as they can be encoded using typeattributes.

Sarna-Starosta et al. [44] propose a logic-programming based approach to analyzing SELinux policies. Their tool transforms a configuration into a Prolog program, thus allowing administrators to perform deductions on configuration properties with the standard Prolog query mechanism. This proposal is similar to ours except that we target CIL and allow labels inside configurations. Also, they rely on libraries of predefined queries for assisting users not familiar with logic programming. Our DSL precisely targets information flows, and easily compiles into LTL.

Hernandez et al. [25] investigate the Android policy environment, which includes discretionary access control common to UNIX, mandatory access control via SElinux, as well as Linux capabilities. The authors propose BigMAC, a framework that helps in maintaining the integrity of an Android system by combining and instantiating all layers of the policy together and producing a fine grained, very large attack graph for whole system. A logic-based query engine is then used to filter out paths and types not in use. The framework detected multiple policy issues on Samsung S8+ and LG G7. Similarly, Lee et al. [29] describe PolyScope, a tool to analyze Android systems and identify the resources that attackers can modify, with and without policy manipulations. Moreover, for each integrity violation detected, PolyScope determines how adversaries may launch attacks to compromise the vulnerable resources. PolyScope has been applied to some Google and OEM Android releases, finding new vulnerabilities and showing that permission expansion increases the privileges available to launch attacks. Unlike these papers, we only focus on SELinux configurations and directly target the policy language. We also express transitive and non-transitive information flow properties, besides reachability studied there. For example, proving that every information flow passes through a selected node is easily doable in our framework (e.g., the compartmentalization and the CW-lite properties in Table 1). However, it is neither expressible in [25] nor through the queries predefined by [29].

Shankar et al. [45] propose CW-Lite, a lighter version of the Clark-Wilson integrity model [13], still guaranteeing that an information flow from a untrusted to a critical node can occur if sanitized by passing through a special trusted node. CW-Lite applies user defined filtering only to those input interfaces deemed untrusted by the policy, and its verification requires annotating applications. A verifier for CW-Lite integrity is included in the SELinux user library and kernel, through which integrity-violating configuration errors have been found in the default SELinux policies for OpenSSH and vsftpd. Differently from this approach, we do not require classifying nodes in trusted/untrusted, therefore allowing for a flexible handling of the flows to be checked.

Domain Specific Languages. Several high-level languages have been proposed for SELinux incorporating notions of information flow. All these languages were presented prior to the introduction of CIL; they therefore target the kernel policy language and do not exploit CIL's advanced features. In contrast, we consider an already adopted language, namely CIL, and extend it with useful features, that support administrators in reasoning about their code. Moreover, IFCIL is backward compatible. Administrators thus neither need to change the workflow nor the tools they use to develop and maintain SELinux configurations.

Hurd et al. [26] propose Lobster, a high-level DSL for specifying SELinux configurations. This compositional language describes the configuration's expected information flow. Instead of macros and blocks, Lobster provides the user with class definition and instantiation, where operations and permissions are represented as ports and labeled arrows between ports, respectively. The user must specify all the desired information flows of the system and the compiler checks that no others are possible. In contrast, we allow the user to succinctly specify wanted and unwanted information flows. In particular, user can also specify "negative" requirements that explicitly forbid some information flows, while Lobster allows specifying only the "positive" flows. Moreover, IFCIL supports more fine-grained requirements, letting users choose the level of details in defining the information flow in the system, e.g., targeting only critical permissions. Finally, Lobster is not backward compatible with SELinux, whereas IFCIL configurations are legal CIL configurations.

Nakamura et al. [33] propose SEEdit, a security policy configuration system that supports creating SELinux configurations using a high-level language called the Simplified Policy Description Language (SPDL). SPDL keeps the configuration small because the administrator can group SELinux permissions and refer to system resources directly using their name instead of types. They implement a converter that produces SELinux configurations, and they propose a set of tools for automatically deriving (parts of) a configuration using system logs. Their main objective is mainly to simplify the usage of the kernel policy language, working on its syntax and adding utility features. Static checking is not supported.

Information flow in access control. Bugliesi et al. [3] develop a verification framework supported by a tool for `grsecurity`, a role-based access control system for Unix/Linux [48]. They propose an operational semantics for `grsecurity` and an abstraction mechanism that reduces the problem of policy verification to reachability, thereby allowing for model checking. The properties they address concern establishing whether a given subject can access a given resource and the writing and reading flows on resources. Although they address properties similar to ours, these properties are built into the verification framework and there is no language for formalizing new security requirements. Moreover, they do not address non-transitive properties as we do.

Calzavara et al. [5] continue this line of research by proposing a security type system for verifying information flow in ARBAC policies. In particular, they abstractly model policies as transition systems and address the role reachability problem with a compositional technique. Also these policies are built into the type system and do not cover non-transitive properties. In contrast, we use a suitable extension of CIL to express policies, we handle non-transitive properties, and we combine different IFCIL policies, as it is common for CIL. Since we use model checking, the results of our verification phase cannot, however, be composed.

Guttman and Herzog [23] consider a network access control scenario. They propose a formalism for expressing networks and security goals about the trajectories a network packet can follow. Moreover, they propose ad hoc algorithms that determine if the security goals are satisfied by a system. Their security goals are similar to our information flow requirements in terms of the expressible properties. However, we reduce verification to standard model checking.

Adding information flow to programming languages. Numerous papers address the control of information flow in the language-based approach where programmers specify how data may be used. Below we consider some proposals, focussing on full-fledged security-typed languages.

FlowCAML [36] is a variant of ML with information-flow types and type inference and provides support for the static enforcement of Denning-style confidentiality policies.

Jif [32] extends Java with the decentralized label model where data values are labeled with security policies. The Jif compiler enforces these security policies performing some static checking. Moreover, Jif supports declassification, which provides a liberal information flow escape hatch for programs that would otherwise be rejected by the compiler.

Fabric [30] extends Jif with support for distributed programming and transactions. It provides several mechanisms for controlling accesses and information flow, to prevent violating confidentiality and integrity policies. All values in Fabric are labeled with policies in the decentralized label model that express security requirements in terms of principals. These labels allow principals to control to what extent other principals can learn or affect their information.

Lifty [35] is a domain-specific language for data-centric applications that allows programmers to annotate the sources of sensitive data with declarative information flow policies. Lifty uses liquid types to enforce static information flow control and to statically and automatically verify that the application obeys the policies. Moreover, its compiler is equipped with a repair engine that automatically patches any found leaks.

Paragon [2] extends Java with information flow policies building on an object-oriented generalisation of Paralocks [1]. A policy is a set of flow locks that are conditions constraining how principals handle data and that can be opened and closed via instructions. Paragon expresses a wide variety of policy paradigms, including Denning-style policies, the Jif decentralised label model, and stateful information flow policies.

All the above papers address Turing complete languages. They encode policies through security labels and enforce them through security type systems. We instead propose a declarative language for expressing policies that describe the admitted and prohibited flows rather than associating labels to resources and user. Also, we target a configuration language that is not Turing complete, and our verification mechanism is based on model checking. Finally, in contrast to some of the above proposals, we do not explicitly deal with declassification.

8 Conclusions and future work

We have proposed IFL, a language for expressing fine-grained information flow requirements. Its declarative nature makes it easy to embed it in various access control languages and facilitates requirement verification through standard model checkers. We exploit IFL to obtain IFCIL, a backwards compatible extension of CIL. IFCIL helps administrators in writing information flow policies, including confidentiality and integrity requirements. We have also defined and implemented a verification procedure to check if an IFCIL configuration complies with its IFL requirements. Our experiments show that the language works well for defining properties that are commonly investigated for SELinux policies, and that the verification times are acceptable even for large real-world configurations.

Discussion. We believe that our extension can help the development of more advanced high-level languages. As our annotations are associated with a common intermediate language, they can enrich different high-level languages. Our verification procedure can be used for checking properties when composing code written in different languages.

Our semantics focuses on CIL type enforcement because it allows defining more fine-grained information flow policies than other constructs, like those for multi-level security [40]. Moreover, many real CIL configurations only use these more limited constructs. We do not explicitly model the constructs for defining the operations used inside allow rules. But this is not a limitation because these constructs can be easily encoded in the considered fragment. Indeed, as we discussed in Section 6.2, our tool deals with all the type enforcement constructs used in real-world CIL configurations.

Our extension targets well known problems in policy development. Moreover, it provides a basis for developing and implementing new high-level languages for SELinux as our semantics completes the existing, informal, and incomplete, CIL documentation. Our proposal can also be applied to check properties when composing code written in different high-level languages sharing this common intermediate language.

Since the actual SELinux architecture uses CIL as an intermediate language, our tool can also be used to verify properties of configurations written in the current policy language. This includes the SELinux reference policy that is part of several Linux distributions, and the Android policy [18].

Future work. There are several exciting directions for future work that aim at fostering the adoption of IFCIL by practitioners. First, we plan to cover all the features of the CIL language, even though the type enforcement fragment that we currently support suffices to analyze many real-world configurations. We will also provide more friendly diagnostics and suggestions for fixing violated requirements.

We plan to enhance our tool's efficiency by reengineering and optimizing its code. Also we will address the issues of modular and incremental analysis. We consider these aspects critical for the integration of IFCIL in the life-cycle of CIL configurations. In particular, we aim at supporting the development of tools like IDEs that provide instant feedback to administrators while they are writing their configurations, as is sometimes done with typed languages.

Finally, we plan to support configurations partly written in the kernel policy language and partly written in CIL, as this is common practice [19].

ACKNOWLEDGEMENTS

Work partially supported by projects SERICS (PE00000014) and PRIN AM∀DEUS (P2022EPPHM) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Niklas Broberg and David Sands. 2010. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th POPL*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 431–444.

[2] Niklas Broberg, Bart van Delft, and David Sands. 2017. Paragon - Practical programming with information flow control. *J. Comput. Secur.* 25, 4-5 (2017), 323–365.

[3] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Marco Squarcina. 2012. Gran: Model Checking Grsecurity RBAC Policies. In *25th IEEE Computer Security Foundations Symposium,*, Stephen Chong (Ed.). IEEE Computer Society, 126–138.

[4] Daniel Burgener. 2024. *Cascade: A high level language for SELinux policy.*

[5] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2015. Compositional Typed Analysis of ARBAC Policies. In *IEEE 28th Computer Security Foundations Symposium*, Cédric Fournet, Michael W. Hicks, and Luca Viganò (Eds.). 33–45.

[6] James Carter. 2024. *[PATCH 1/3] libsepol/cil: Make name resolution in macros work as documented.*

[7] Lorenzo Ceragioli. 2024. *Bug (?) report for secilc and CIL semantics: some unexpected behaviours.*

[8] Lorenzo Ceragioli. 2024. *[bug report?] other unexpected behaviours in secilc and CIL semantics.*

[9] Lorenzo Ceragioli. 2024. *SELinux CIL Semantics.*

[10] Lorenzo Ceragioli. 2024. *SELinux IFCIL tool.*

[11] Lorenzo Ceragioli, Letterio Galletta, Pierpaolo Degano, and David A. Basin. 2022. IFCIL: An Information Flow Configuration Language for SELinux. In *CSF*. IEEE, 243–259.

[12] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *14th Computer Aided Verification, Copenhagen, DK, 2002, (LNCS, Vol. 2404)*, E. Brinksma and K. G. Larsen (Eds.). Springer, 359–364.

[13] David D. Clark and D. R. Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*. IEEE Computer Society, 184–195. https://doi.org/10.1109/SP.1987.10001

[14] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (may 1976), 236–243.

[15] Florida Institute for Cybersecurity. 2024. *BigMAC: Analysis tool to introspect and query Android security policies*.

[16] Maurizio Gabbrielli and Simone Martini. 2010. *Programming Languages: Principles and Paradigms*. Springer. https://doi.org/10.1007/978-1-84882-914-5

[17] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11.

[18] Google. [n. d.]. *Android Open Source Project*.

[19] Google. 2024. *sepolicy*.

[20] Dominick Grift. 2024. *openWRT SELinux policy (commit aa59f95 on 3 Dec 2021)*.

[21] Dominick Grift. 2024. *SELinux example policy cilpolicy (commit 562a8a2 on 8 Sep 2015)*.

[22] Dominick Grift. 2024. *SELinux example policy dspp5 (commit cf4dd60 on 16 Dec 2021)*.

[23] Joshua D. Guttman and Amy L. Herzog. 2005. Rigorous automated network security management. *Int. J. Inf. Sec.* 4, 1-2 (2005), 29–48.

[24] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. 2005. Verifying information flow goals in Security-Enhanced Linux. *J. Comput. Secur.* 13, 1 (2005), 115–134.

[25] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. 2020. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 271–287. https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

[26] Joe Hurd, Magnus Carlsson, Brett Letner, and Peter White. 2008. *Lobster: A domain specific language for selinux policies*. Technical Report. Galois, Inc.

[27] Bumjin Im, Ang Chen, and Dan S. Wallach. 2018. An Historical Analysis of the SEAndroid Policy Evolution. In *Procs. 34th Annual Computer Security Applications Conference*. ACM, 629–640.

[28] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. 2003. Analyzing Integrity Protection in the SELinux Example Policy. In *Procs 12th USENIX Security Symposium, Washington, D.C., USA, 2003*. USENIX Association.

[29] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, Giuseppe Petracca, and Trent Jaeger. 2021. PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2579–2596. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yu-tsung

[30] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. 2017. Fabric: Building open distributed systems securely by construction. *J. Comput. Secur.* 25, 4-5 (2017), 367–426.

[31] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. 1999. Model-Checking: A Tutorial Introduction. In *Procs 6th Static Analysis Symposium, LNCS,1694*, A. Cortesi and G. Filé (Eds.). Springer, 330–354.

[32] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 228–241.

[33] Yuichi Nakamura, Yoshiki Sameshima, and Toshihiro Yamauchi. 2010. SELinux Security Policy Configuration System with Higher Level Language. *J. Inf. Process.* 18 (2010), 201–212.

[34] OpenWRT. 2024. .

[35] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *Proc. ACM Program. Lang.* 4, ICFP, Article 105 (aug 2020), 30 pages.

[36] François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1 (jan 2003), 117–158.

[37] SELinux project. 2024. *SELinux project*.

[38] SELinux Project. 2024. *The SELinux Notebook - CIL Reference Guide (commit 2b3b4ee on 6 Jul 2020)*.

[39] SELinux Project. 2024. *The SELinux Notebook - Kernel Policy Language*.

[40] SELinux Project. 2024. *The SELinux Notebook - MLS Statements*.

[41] B. S. Radhika, N. V. Narendra Kumar, R. K. Shyamasundar, and Parjanya Vyas. 2020. Consistency analysis and flow secure enforcement of SELinux policies. *Comput. Secur.* 94 (2020), 101816.

[42] Elena Reshetova, Filippo Bonazzi, and N. Asokan. 2017. SELint: An SEAndroid Policy Analysis Tool. In *Procs 3rd International Conference on Information Systems Security and Privacy, Porto, PT, 2017*, P. Mori, S. Furnell, and O. Camp (Eds.). SciTePress, 47–58.

[43] A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[44] Beata Sarna-Starosta and Scott D. Stoller. 2004. Policy Analysis for Security-Enhanced Linux. In *Procs 2004 Workshop on Issues in the Theory of Security*. 1–12.

[45] Umesh Shankar, Trent Jaeger, and Reiner Sailer. 2006. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2006/toward-automated-information-flow-integrity-verification-security-critical-applications/

[46] Stephen Smalley. 2018. *Add support for a source policy HLL.*

[47] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *20th Annual Network and Distributed System Security Symposium, 2013, San Diego, California, USA, February 24-27, 2013.* The Internet Society.

[48] Bradley Spengler. 2004. *Increasing performance and granularity in rolebased access control systems.*

[49] Dan Thomsen. 2004. *Information Flow Analysis in Security Enhanced Linux.*

[50] Toshihiro Yokoyama, Miyuki Hanaoka, Makoto Shimamura, Kenji Kono, and Takahiro Shinagawa. 2009. Reducing Security Policy Size for Internet Servers in Secure Operating Systems. *IEICE Trans. Inf. Syst.* 92-D, 11 (2009), 2196–2206.

A Technical notes on the CIL formalization

Given a configuration R, we build its *initial environment* ρ_R by associating every qualified name in a namespace with its value.

$$\begin{aligned}
\rho_R &= initial\_env(R, \#)\\
initial\_env(R, ns) &= \bigcup_{rule \in R} initial\_env(rule, ns)\\
initial\_env((\texttt{type}\ dn), ns) &= (ns.dn, ns.dn)\\
initial\_env((\texttt{typeattribute}\ dn), ns) &= (ns.dn, ns.dn)\\
initial\_env((\texttt{macro}\ dn\ (\widetilde{x})\ M), ns) &= (ns.dn, \widetilde{x}, M, [ns])\\
initial\_env((\texttt{block}\ dn\ R), ns) &= (ns.dn, R) \cup initial\_env(R, ns.dn)
\end{aligned}$$

We let $\rho[ns \oplus fr]$ be syntactic sugar for $\rho[ns \mapsto fr \cup \rho(ns)]$, where the bindings of fr are added to the ones of the frame $\rho(ns)$ (instead of substituted).

Given a ruleset R, an environment ρ and a context σ, we constrain all $\mathcal{T}_i$ to be such that

$$\mathcal{T}_i(R, \rho, \sigma) = \bigcup_{rule \in R} \mathcal{T}_i(rule, \rho, \sigma)$$

Given a ruleset R, an environment ρ and a context σ, the first function $\mathcal{T}_1$ returns a new ruleset R' where globally qualified names of the blocks replace qualified names in the blockinherit rules:

$$\begin{aligned}
\mathcal{T}_1((\textbf{block}\ dn\ R),\ \rho, [ns \mid \sigma]) &= (\textbf{block}\ dn\ R') && \text{with } R' = \mathcal{T}_1(R,\ \rho, [ns.dn \mid \sigma])\\
\mathcal{T}_1((\textbf{blockinherit}\ qn), \rho, \sigma) &= \{(\textbf{blockinherit}\ gn)\} && \text{where } \bar{\rho}(\sigma)(qn) = (gn, R)\\
\mathcal{T}_1(rule, \rho\ \sigma) &= \{rule\} && \text{if } rule \text{ is not } block \text{ or } blockinherit
\end{aligned}$$

After $\mathcal{T}_1$, we build ρ again by inspecting the rules, so that the names in blockinherit statements of the ruleset matches the ones in the statements inside ρ.

The second transformation $\mathcal{T}_2$ updates the environment ρ by propagating the declarations from inherited to inheriting blocks. We need an auxiliary function that, given a frame, updates the globally qualified names of the entities it contains and the clousure of macros with a given namepace. Formally:

$$\begin{aligned}
upd(ns.dn, ns') &= ns'.dn && \text{for types and typeattributes}\\
upd((ns.dn, R), ns') &= (ns'.dn, R) && \text{for blocks}\\
upd((ns.dn, pars, M, \xi, \sigma), ns') &= (ns'.dn, pars, M, \xi, [ns' \mid \sigma]) && \text{for macros}\\
upd(fr, ns) &= \bigcup_{(dn,val)\in fr} (dn, upd(val, ns)) && \text{for frames}
\end{aligned}$$

We can now define $\mathcal{T}_2$ as follows:

$$\begin{aligned}
\mathcal{T}_2((\textbf{block}\ dn\ R),\ \rho, [ns \mid \sigma]) &= \mathcal{T}_2(R,\ \rho, [ns.dn \mid \sigma])\\
\mathcal{T}_2((\textbf{blockinherit}\ gn), \rho, [ns \mid \sigma]) &= \rho[ns \oplus upd(\rho'(gn), ns)]\\
&\qquad \text{where } \rho' = \mathcal{T}_2(R, \rho, [ns \mid \sigma]) \text{ and } \overline{\rho}([\,])(gn) = (gn, R)\\
\mathcal{T}_2(rule, \rho, \sigma) &= \rho \qquad \text{if } rule \neq block, blockinherit
\end{aligned}$$

Finally, the third transformation $\mathcal{T}_3$ propagates the declarations by copying the names defined in a macro inside its caller's namespace. In the definition of $\mathcal{T}_3$ we use the auxiliary function $cdm(m, \rho, \sigma)$ that collects the names of the types and of the typeattributes declared in the given macro m and in those it calls. It is inductively defined as follows:

$$cdm(m, \rho, \sigma) = \xi \cup \bigcup_{(\textbf{call}\ m'(\widetilde{qn}))\in M} cdm(m', \rho, \sigma' \cdot \sigma) \qquad \text{where } (gn, \widetilde{x}, M, \xi, \sigma') = \overline{\rho}(\sigma)(m)$$

The definition of the transformation $\mathcal{T}_3$ follows:

$$\begin{aligned}
\mathcal{T}_3((\textbf{call}\ m\ (\widetilde{qn'})), \rho, [ns \mid \sigma]) &= \rho[ns \oplus \{(dn, ns.dn) \mid dn \in cdm(m, \rho, \sigma)\}]\\
\mathcal{T}_3((\textbf{blockinherit}\ ns.dn), \rho, [ns' \mid \sigma]) &= \rho[ns' \oplus \rho'(ns.dn)]\\
&\qquad \text{where } \rho' = \mathcal{T}_3(R, \rho, [ns' \mid \sigma \cdot [ns]]) \text{ and } \overline{\rho}(ns)(dn) = (ns.dn, R)\\
\mathcal{T}_3(rule, \rho, \sigma) &= \rho \qquad \text{if } rule \text{ is not } block, call, blockinherit
\end{aligned}$$

B Proofs for IFL refinement

Lemma B.1 (Kind refinement). *Let I be an information flow diagram and π a path in I, if $P \preceq_P P'$ and $\pi \triangleright_I P$ then $\pi \triangleright_I P'$.*

Proof. Assume $\pi \triangleright_I P$ and not $\pi \triangleright_I P'$ and proceed by proving for each rule in Figure 7 and for arbirary π that if the lemma holds on the premises, that it also does on the conclusion.

Consider the rule (comp), and let $i \in \{1, 2\}$. From $\pi_i \triangleright_I P_i$ we have that $\pi_i \triangleright_I P'_i$. Then $\pi_1\pi_2 \triangleright_I P'_1P'_2$ by the definition of $\triangleright$.

Consider the rule (P-1) and proceed by cases on the form of the arrow in oa. If the arrow is $>$ then from $\pi \triangleright_I (n_1[o] > n_2)$ we know that $\pi = (n''_1, o'', n''_2)$ (since this is the only possible case in the definition of $\triangleright$). We also know that $n''_1 \in ta(n_1)$ or $n_1 = *$, but since $n_1 \preceq n'_1$ either $n_1 = n'_1$ or $n'_1 = *$, and then $n''_1 \in ta(n'_1)$ or $n'_1 = *$. Similarly for n''_2. Finally, from the definition of $\triangleright$ we also know that $o \cap o'' \neq \emptyset$, but since $o \preceq o'$, $o \subseteq o'$, thus $o \cap o' \neq \emptyset$. All the requirements for $\pi \triangleright_I (n'_1[o'] > n'_2)$ are thus verified. If the arrow is $+>$, two rules in the definition of $\triangleright$ may be used. The first one has same condition of the previous case, and the second one requires induction on π (note that $* \preceq *$).

Consider the rule (P-2) and assume $\pi \triangleright_I n_1\ +[o_1] >\ *\ [o_2] > n_2$. Then, by the definition of $\triangleright$ we have that $\pi = \pi'(n, o, n''_2)$ with $\pi' \triangleright_I n_1\ +[o_1] >\ *$; $n''_2 \in ta(n_2)$ if $n_2 \neq *$, and $o \cap o_2 \neq \emptyset$. From $\pi' \triangleright_I n_1\ +[o_1] >\ *$ it follows that either $\pi' = (n''_1, o', n')$ or $\pi' = (n''_1, o', n')\pi''$; in both cases $n''_1 \in ta(n_1)$ if $n_1 \neq *$, and $o' \cap o_1 \neq \emptyset$. If $\pi' = (n''_1, o', n')$ then the thesis holds since $\pi' \triangleright_I n'_1\ [o'_1] >\ *$ and $(n, o, n''_2) \triangleright_I *[o'_2] > n'_2$ (thus $(n, o, n''_2) \triangleright_I * + [o'_2] > n'_2$ by the definition of $\triangleright$).

If $\pi' = (n_1'', o', n')\pi''$, then from $\pi' \triangleright_I n_1\ +[o_1] >\ *$ and all (n'', o'', n''') in π', $o'' \cap o_1 \neq \emptyset$ holds. The thesis follows, since $o_1 \preceq o_2'$, $\pi''(n, o, n_2'') \triangleright_I * + [o_2'] > n_2'$.

Consider the rule (P-3) and assume $\pi \triangleright_I n_1\ +[o_1] >\ *\ +[o_2] > n_2$. Then split π as $\pi'\pi''$ such that $\pi' \triangleright_I n_1\ +[o_1] >\ *$ and $\pi'' \triangleright_I *\ +[o_2] > n_2$. From $\pi' \triangleright_I n_1\ +[o_1] >\ *$, the first element of π' is some (n, o, n') such that $n \in ta(n_1)$ if $n_1 \neq *$ and thus that $n \in ta(n_1')$ if $n_1' \neq *$. Moreover, for every (n_1'', o_1', n_2'') in π', $o_1' \cap o_1 \neq \emptyset$. From $\pi'' \triangleright_*\ +[o_1] >\ n_2$, the last element of π' is some (n'', o'', n''') such that $n''' \in ta(n_2)$ if $n_2 \neq *$ and thus $n \in ta(n_1')$ if $n_1' \neq *$. Moreover, for every (n_1''', o_2', n_2''') in π'', $o_2' \cap o_2 \neq \emptyset$. Finally, since $o_1 \preceq o'$, and $o_2 \preceq o'$, for every $(\_, o''', \_)$ in π, $o''' \cap o' \neq \emptyset$.

The proof of rule (P-4) is similar to that of (P-2). □

Theorem 4.4. *Let I be an information flow diagram, and let $\mathcal{R}'$ and $\mathcal{R}$ be two IFL requirements such that $\mathcal{R}' \preceq \mathcal{R}$. Then $I \models \mathcal{R}'$ implies $I \models \mathcal{R}$.*

Proof. We proceed by rule induction. Consider the rule ($\mathcal{R}$-1) and assume that $I \models P$, then there exists a path π in I such that $\pi \triangleright_I P$. Since $P \preceq P'$, $\pi \triangleright_I P'$ by Lemma B.1, thus $I \models P'$.

Consider the rule ($\mathcal{R}$-2), and assume that $I \models {\sim}P'$. We proceed by contradiction, assuming $I \not\models {\sim}P$ and showing a contradiction. By the definition, $I \not\models {\sim}P$ implies $I \models P$ and thus there exists a path π in I such that $\pi \triangleright_I P$. Since $P \preceq P'$, $\pi \triangleright_I P'$ by Lemma B.1 and thus $I \models P'$, i.e., $I \not\models {\sim}P'$.

Consider the rule ($\mathcal{R}$-3), and assume that $I \models P_1' : P_2$. Assume now by contradiction that $I \not\models P_1 : P_2'$, which implies that there exists a path π in I such that $\pi \triangleright_I P_1$ and not $\pi \triangleright_I P_2'$. Since $P_1 \preceq P_1'$, $\pi \triangleright_I P_1'$ by Lemma B.1. Thus, since $I \models P_1' : P_2$, $\pi \triangleright_I P_2$ must hold, but then $\pi \triangleright_I P_2'$ must also hold since $P_2 \preceq P_2'$. □

C Technical notes and proofs for IFCIL verification

It is convenient to use in the following the grammar of P of Section 5, and to redefine the relation $\triangleright_I$ of Definition 4.1 in the following, trivially equivalent way.

$$\begin{aligned}
(\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}' &\text{ if } (m = * \ \vee\ \mathtt{n} \in ta(m))\ \wedge (m' = * \ \vee\ \mathtt{n}' \in ta(m'))\ \wedge o \cap o' \neq \emptyset \\
(\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}' &\text{ if } (\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}' \\
(\mathtt{n}, o, \mathtt{n}')\, \pi \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}' &\text{ if } (\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>} * \ \wedge\ \pi \triangleright_I * \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}' \\
(\mathtt{n}, o, \mathtt{n}')\, \pi \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}'\ P &\text{ if } (\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>} * \ \wedge\ \pi \triangleright_I P \\
\pi \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}'\ P &\text{ if } \pi \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}'\ P \\
(\mathtt{n}, o, \mathtt{n}')\, \pi \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}'\ P &\text{ if } (\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>} * \ \wedge\ \pi \triangleright_I * \mathtt{+[}o'\mathtt{]>} * P
\end{aligned}$$

Given a path $\pi = (n_0, o_0, n_1)(n_1, o_1, n_2) \dots (n_{m-1}, o_{m-1}, n_m)$ of an information diagram I with K its KTS, let $\llparenthesis \pi \rrparenthesis$ be the set of paths $w = n_0, op_0, n_1, op_1, n_2, \dots, n_{m-1}, op_{m-1}, n_m$ in K such that $\forall i : op_i \in o_i$. Moreover, given a path of K $w = n_0, op_0, n_1, op_1, n_2, \dots, n_{m-1}, op_{m-1}, n_m$, let π_w be $(n_0, \{op_0\}, n_1)(n_1, \{op_1\}, n_2) \dots (n_{m-1}, \{op_{m-1}\}, n_m)$ of I.

Lemma C.1. *Let π be a path in the information diagram I with K its KTS, and let P be an information flow kind. Then $\pi \triangleright_I P$ if and only if $w \models_l \llparenthesis P \rrparenthesis$ for some $w \in \llparenthesis \pi \rrparenthesis$.*

Proof. We proceed by induction on P. For P of the form $\mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}'$ or $\mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}'$ the properties trivially hold. For the last two cases of P we proceed as follows.

Case $P = \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}'\ P'$. Assume $\pi \triangleright_I P$. By the definition of $\triangleright_I$, $\pi = (n, o, n')\ \pi'$, with $o \cap o' \neq \emptyset$. We also know by the induction hypothesis that there exists $w' \in \llparenthesis \pi' \rrparenthesis$ such that $w' \models_l \llparenthesis P' \rrparenthesis$. Since $o \cap o' \neq \emptyset$, there exists $op \in o \cap o'$. Take $w = n, op, w'$. Then $w \in \llparenthesis \pi \rrparenthesis$ holds by definition, and also $w \models_l \llparenthesis P \rrparenthesis$.

Assume $w \models_l \llparenthesis P \rrparenthesis$. Since $\llparenthesis P \rrparenthesis = m \land \bigvee_{op \in o'}(op) \ \land X(\llparenthesis P' \rrparenthesis)$, $w = n, op, w'$ with $n \in ta(m)$ and $op \in o'$ trivially follows from the semantics of LTL. Then $\pi = (n, o, n')\pi'$ with $op \in o$ and $w' \in \llparenthesis \pi' \rrparenthesis$. We also know by the induction hypothesis that $\pi' \triangleright_I P'$, then $\pi \triangleright_I P$.

Case $P = \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m'}\ P'$. Assume $\pi \triangleright_I P$. We separately consider the two cases of the definition of $\triangleright$ that apply to P. If $\pi \triangleright_I \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m'}\ P'$, then we are in the previous case. Otherwise, $\pi = (n, o, n')\ \pi'$ with $(n, o, n') \triangleright_I m\mathtt{[}o'\mathtt{]>}{*}$ and $\pi' \triangleright_I {*}\mathtt{[}o'\mathtt{]>}\ {*}\ P'$. By induction hypothesis, $w' \in \llparenthesis \pi' \rrparenthesis$ exists such that $w' \models_l \bigvee_{op \in o'}(op) \ \land X(\bigvee_{op \in o'}(op)\ U\ \llparenthesis P' \rrparenthesis)$. Thus, $w = n, op, w'$ exists such that $op \in o$, and it holds that $w \in \llparenthesis \pi \rrparenthesis$. The thesis follows from the following stronger statement that trivially holds by construction: $w \models_l m \land \bigvee_{op \in o'}(op) \ \land X(\bigvee_{op \in o'}(op) \land X(\bigvee_{op \in o'}(op)\ U\ \llparenthesis P' \rrparenthesis))$.

Assume $w \models_l \llparenthesis P \rrparenthesis$. Then either $w \models_l \bigvee_{op \in o'}(op) \ \land X(\llparenthesis P' \rrparenthesis)$ or $w \models_l \bigvee_{op \in o'}(op) \ \land X(\bigvee_{op \in o'}(op) \ \land X(\bigvee_{op \in o'}(op)\ U\ \llparenthesis P' \rrparenthesis))$. If $w \models_l \bigvee_{op \in o'}(op) \ \land X(\llparenthesis P' \rrparenthesis)$, then we are in the previous case. Consider now the second case, and let w be n, op, w'. By definition, $\pi = (n, o, n')\pi'$, with $op \in o$ and $w' \in \llparenthesis \pi' \rrparenthesis$. By construction, $(n, o, n') \triangleright_I m\mathtt{[}o'\mathtt{]>}{*}$, and by induction hypothesis, $\pi' \triangleright_I {*}\mathtt{+[}o'\mathtt{]>}\ {*}\ P$. □

Lemma C.2. *Let w be a path in the KTS K of the information diagram I, and let P be an information flow kind. Then $w \models_l \llparenthesis P \rrparenthesis$ if and only if $\pi_w \triangleright_I P$.*

Proof. Trivially derives from Lemma C.1 since $\llparenthesis \pi_w \rrparenthesis = \{w\}$. □

Theorem 5.4. *Let Σ be an IFCIL configuration with requirements $\mathbb{R}$, let I be its information flow diagram, and let K be the KTS of Σ. Then $K \vdash \mathbb{R}$ if and only if $I \models \mathbb{R}$.*

Proof. We proceed by cases on $\mathcal{R}$.

Case $\mathcal{R} = P$. Assume $I \models P$. Then there exists π such that $\pi \triangleright_I P$, and by Lemma C.1 $K \vdash P$. Conversely, assume $K \vdash P$, thus there exists w such that $w \models_l \llparenthesis P \rrparenthesis$, and by Lemma C.2 $I \models P$.

Case $\mathcal{R} = {\sim}P$. Assume $I \models {\sim}P$. Then there exists no π such that $\pi \triangleright_I P$. Assume by contradiction that $K \nvdash P$. Then $w \models_l \llparenthesis P \rrparenthesis$ for some w, and by Lemma C.2 $\pi_w \triangleright_I P$. Contradiction.

Conversely, assume $K \vdash {\sim}P$, thus there exists no w such that $w \models_l \llparenthesis P \rrparenthesis$. Assume by contradiction that $I \not\models {\sim}P$. Then there exists π such that $\pi \triangleright_I P$ and, by Lemma C.2 there exists w such that $w \models_l \llparenthesis P \rrparenthesis$. Contradiction.

Case $\mathcal{R} = P : P'$. Assume $I \models P : P'$. Then there exists no π such that $\pi \triangleright_I P$ and $\pi \not\triangleright_I P$. Assume by contradiction that $K \nvdash P : P'$. Then there exists w such that $w \models_l \llparenthesis P \rrparenthesis$ and $w \not\models_l \llparenthesis P' \rrparenthesis$, and by Lemma C.2 $\pi_w \triangleright_I P$ and $\pi_w \not\triangleright_I P$. Contradiction.

Conversely, assume $K \vdash P : P'$, thus there exists no w such that $w \models_l \llparenthesis P \rrparenthesis$ and $w \not\models_l \llparenthesis P' \rrparenthesis$. Assume by contradiction that $I \not\models P : P'$. Then there exists π such that $\pi \triangleright_I P$ and $\pi \not\triangleright_I P'$. Thus, by Lemma C.1, there exists w such that $w \models_l \llparenthesis P \rrparenthesis$ and $w \not\models_l \llparenthesis P' \rrparenthesis$. Contradiction. □

In the following we write $\bar{W}$ and $\bar{W}_\iota$ for infinite paths in K and K_ι; $\dot{W}$ and $\dot{W}_\iota$ for finite paths in K and K_ι; and W_ι for $\bar{W}_\iota \cup \dot{W}_\iota$ (note that $W = \bar{W} \cup \dot{W}$). With a small abuse of notation, we write $K_\iota \models_l \phi$ if and only if $\forall w \in \bar{W}_\iota . w \models_l \phi$.

Definition C.3. The encoding of flow kinds for K_ι is as follows.

$$\llparenthesis \mathtt{n}\ \mathtt{[o]>}\ \mathtt{n'} \rrparenthesis_\iota = \mathtt{n} \land \bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}) \land X(\mathtt{n'} \land X(\iota))$$

$$\llparenthesis \mathtt{n}\ \mathtt{+[o]>}\ \mathtt{n'} \rrparenthesis_\iota = \mathtt{n} \land \bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}) \land X(\bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op})\ U\ (\mathtt{n'} \land X(\iota)))$$

$$\llparenthesis (\mathtt{n}\ \mathtt{[o]>}\ \mathtt{n'})P \rrparenthesis_\iota = \mathtt{n} \land \bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}) \land X(\llparenthesis P \rrparenthesis_\iota)$$

$$\llparenthesis(\mathtt{n}\ \mathtt{+[o]>}\ \mathtt{n}')P\rrparenthesis_\iota = \mathtt{n} \land \bigvee_{\mathtt{op}\in\mathtt{o}} (\mathtt{op}) \land X(\bigvee_{\mathtt{op}\in\mathtt{o}} (\mathtt{op})\ U\ \llparenthesis P\rrparenthesis_\iota)$$

The satisfaction relation $\vdash_\iota$ is defined as follows.

$$K_\iota \vdash_\iota P \text{ iff } K_\iota \not\models_l \neg\llparenthesis P\rrparenthesis_\iota$$
$$K_\iota \vdash_\iota {\sim}P \text{ iff } K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota$$
$$K_\iota \vdash_\iota P{:}P' \text{ iff } K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota \lor \llparenthesis P'\rrparenthesis_\iota$$

Lemma C.4. *If $w \models_l \llparenthesis P\rrparenthesis$ then $w \in \dot{W}$ and $w\iota^\omega \models_l \llparenthesis P\rrparenthesis_\iota$ with $w\iota^\omega \in \bar{W}_\iota$.*

Proof. Trivial. □

Lemma C.5. *If $w \models_l \llparenthesis P\rrparenthesis_\iota$ then there exists a unique $\dot{w} \in \dot{W}$ such that $\dot{w} \models_l \llparenthesis P\rrparenthesis$ and $w = \dot{w}\iota^\omega$.*

Proof. Trivial. □

Corollary C.6. *Let Σ be an IFCIL configuration with requirements $\mathbb{R}$, let I be its information flow diagram, and let K be the KTS of Σ. Then $K_\iota \vdash \mathbb{R}$ if and only if $K \vdash \mathbb{R}$ if and only if $I \models \mathbb{R}$.*

Proof. It suffices to prove the following.

$$K \models_l \neg\llparenthesis P\rrparenthesis \text{ iff } K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota \tag{7}$$
$$K \models_l \neg\llparenthesis P\rrparenthesis \lor \llparenthesis P'\rrparenthesis \text{ iff } K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota \lor \llparenthesis P'\rrparenthesis_\iota \tag{8}$$

Proof of (7). Assume by contradiction that $K \models_l \neg\llparenthesis P\rrparenthesis$ and $K_\iota \not\models_l \neg\llparenthesis P\rrparenthesis_\iota$. From $K_\iota \not\models_l \neg\llparenthesis P\rrparenthesis_\iota$ we know that there exists $w' \in \bar{W}_\iota$ such that $w' \models_l \llparenthesis P\rrparenthesis_\iota$. By Lemma C.5, there exists $w'' \in W$ such that $w'' \models_l \llparenthesis P\rrparenthesis$. But from $K \models_l \neg\llparenthesis P\rrparenthesis$ we know that $\forall w \in W. w \not\models_l \llparenthesis P\rrparenthesis$. Contradiction.

Assume now by contradiction that $K \not\models_l \neg\llparenthesis P\rrparenthesis$ and $K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota$. From $K \not\models_l \neg\llparenthesis P\rrparenthesis$ we know that there exists $w' \in W$ such that $w \models_l \llparenthesis P\rrparenthesis$. By Lemma C.4, $w'\iota^\omega \in \bar{W}_\iota$ is such that $w'\iota^\omega \models_l \llparenthesis P\rrparenthesis_\iota$. But from $K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota$ we know that $\forall w \in \bar{W}_\iota$ it is $w \not\models_l \llparenthesis P\rrparenthesis$. Contradiction.

Proof of (8). Assume by contradiction that $K \models_l \neg\llparenthesis P\rrparenthesis \lor \llparenthesis P'\rrparenthesis$ and $K_\iota \not\models_l \neg\llparenthesis P\rrparenthesis_\iota \lor \llparenthesis P'\rrparenthesis_\iota$. From $K_\iota \not\models_l \neg\llparenthesis P\rrparenthesis_\iota \lor \llparenthesis P'\rrparenthesis_\iota$ we know that there exists $w' \in \bar{W}_\iota$ such that $w' \models_l \llparenthesis P\rrparenthesis_\iota$ and $w' \not\models_l \llparenthesis P'\rrparenthesis_\iota$. Then, by Lemma C.5, $w' = w''\iota^\omega$ with $w'' \in \dot{W}$ and $w'' \models_l \llparenthesis P\rrparenthesis$. Clearly, $w'' \not\models_l \llparenthesis P'\rrparenthesis$, otherwise we would have that $w' \models_l \llparenthesis P'\rrparenthesis_\iota$ by Lemma C.4. But from $K \models_l \neg\llparenthesis P\rrparenthesis \lor \llparenthesis P'\rrparenthesis$ we know that $\forall w \in W$ it is $w \not\models_l \llparenthesis P\rrparenthesis \lor w \models_l \llparenthesis P'\rrparenthesis$. Contradiction.

Assume now by contradiction that $K \not\models_l \neg\llparenthesis P\rrparenthesis \lor \llparenthesis P'\rrparenthesis$ and $K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota \lor \llparenthesis P'\rrparenthesis_\iota$. From $K \not\models_l \neg\llparenthesis P\rrparenthesis \lor \llparenthesis P'\rrparenthesis$ we know that there exists $w' \in W$ such that $w' \models_l \llparenthesis P\rrparenthesis$ and $w' \not\models_l \llparenthesis P'\rrparenthesis$. Thus, by Lemma C.4, $w'\iota^\omega \models_l \llparenthesis P\rrparenthesis_\iota$ with $w'\iota^\omega \in \bar{W}_\iota$. Clearly, $w'\iota^\omega \not\models_l \llparenthesis P'\rrparenthesis_\iota$, otherwise we would have that $w' \models_l \llparenthesis P'\rrparenthesis$ by Lemma C.4. But from $K_\iota \models_l \neg\llparenthesis P\rrparenthesis_\iota \lor \llparenthesis P'\rrparenthesis_\iota$ we know that $\forall w \in \bar{W}_\iota$ it is $w \not\models_l \llparenthesis P\rrparenthesis_\iota \lor w \models_l \llparenthesis P'\rrparenthesis_\iota$. Contradiction. □