IFCIL: An Information Flow Configuration Language for SELinux

Lorenzo Ceragioli*, Letterio Galletta†§, Pierpaolo Degano*†, and David Basin‡
* Università di Pisa, Italy
† IMT School for Advanced Studies Lucca, Italy
‡ ETH Zurich, Switzerland
§ CINI Cybersecurity National Laboratory, Rome, Italy

***Abstract*—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. But this 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. However, CIL lacks mechanisms for ensuring that the resulting configurations obey desired information flow policies. To remedy this, we propose IFCIL, a backward compatible extension of CIL for specifying fine-grained information flow requirements for CIL configurations. Using IFCIL, administrators can express, e.g., confidentiality, integrity, and non-interference properties. We also provide a tool to statically verify these requirements.**

I. 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 [1], network appliances [2], and mobile devices [3]. 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 [4] that have led to serious vulnerabilities in widely used policies.

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 [5], [6]. 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 indirect 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 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. IFL requirements can be extended through refinement and 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 and empirically validate its adequacy with respect to the CIL reference manual and the CIL compiler. Besides providing the basis for our verification algorithm, the semantics and its experimental validation make it possible to understand CIL's trickier bits, and to illuminate some unspecified corner cases and disagreements between the documentation and the compiler.
- We provide a prototype tool [7] that implements our verification procedure by leveraging NuSMV, a popular model checker [8]. 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 three real-world CIL policies [2], [9], [10]. We annotate them 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 configuration in [2], which has roughly 46,000 lines of code.

Outline: In Section II we introduce SELinux, CIL, and the mechanism used by administrators to protect critical resources. In Section III we give a high-level account of our CIL semantics and how we experimentally validate its adequacy. In Section IV we present IFCIL and we explain our verification procedure for checking the satisfiability of the requirements in Section V. In Section VI we present our verification tool and our experimental assessment. In Section VII we compare our work with the relevant literature and in Section VIII we draw conclusions. The appendices contain the details of our formal development and the proof sketches of our theorems. The full proofs can be found in the extended version of this paper [11].

II. Background

a) SELinux: SELinux is a set of extensions to the Linux kernel and utilities [12]. It extends the major subsystems of the Linux kernel with strong, flexible, mandatory access control (MAC). 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 labelled 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 DAC rights of a directory. A process *P* can invoke a system call *SC* on a resource *R* only if there is a rule that permits *P* to do so.

Administrators typically specify configurations using SELinux's *kernel policy language* [13]. 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 [4]. Some high-level configuration languages have been suggested with their own compilers and tools as an attempt to address these limitations [6], [14]. 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.

b) CIL: The Common Intermediate Language (CIL) [15] 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 [6], and future ones too. Despite its original goal, CIL is also used to directly write configurations [10], [2], [9] for complex real-world policies, like for Android [16]. 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 main difference is that inheritance is actually realized by a kind of copying rule.

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. These examples also illustrate that blocks, types, typeattributes and macros have names, and resolving them in the correct name space and order is non-trivial.

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, and the outermost block can refer to the elements in the nested ones by qualifying their names.

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

Intuitively, the last `allow` rule grants subjects with type `tree.bird` the permission to write to the files with type `tree.nest.egg`.

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

```
(type stranger)
(allow stranger inhouse.object (file (open)))
(block inhouse
    (type man)
    (type object)
    (allow man object (file (read)))
    (allow .stranger object (file (read)))
    (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 however that if the block `inhouse` declared a type `stranger`, this declaration would overshadow the global one in the last `allow` rule, but not the third one since a fully qualified name is used. Note too 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 in the following example.

```
(block animal_mcr
  (macro add_dog((type x)(type y))
    (allow x man (file (read)))
    (allow y dog (file (open))))
  (type dog))
```

Macros are invoked as follows.

```
(block animal_house
  (type man)
  (type cat)
  (call animal_mcr.add_dog(cat cat)))
```

Roughly, the content of `add_dog` replaces the last line where the formal parameters `x` and `y` are bound to `animal_house.cat`. Names are resolved using a mechanism similar to dynamic binding: the name `dog` in the macro is resolved as `animal_mcr.dog`, while `man` is resolved as `animal_house.man`. 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 unexepected behaviour (see Section III 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, which CIL lacks. One contribution of this paper is to provide such a semantics. We define it in Appendix A and provide an intuitive account in Section III.

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

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

The first type attribute includes the two types `animal_mcr.dog` and `animal_house.cat`. 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.

c) An example from the security domain: Consider the following block `mem` defined in [2], a CIL configuration designed for OpenWrt powered wireless routers.

```
(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, 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 the administrator includes 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 vect)
(type deputy)
(typeattributeset mem.read.subj_typeattr deputy)
(allow deputy mem.read.nodedev (chr_file (read)))
(allow deputy vect (file (write)))
(allow untrusted vect (file (read)))
```

where the types `untrusted`, `vect`, and `deputy` are defined, and `deputy` is in `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 `vect`, to an arbitrary process of type `untrusted`, which is permitted by the given `allow` rules.

d) Preventing information flow: Currently, CIL does not prevent indirect information flows between types. 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 behind 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 infomation 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 the pattern 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.

III. Formalizing CIL

The official CIL documentation [15] does not formally describe CIL's syntax and semantics. The following, admittedly artificial, configuration highlights the need for a formal semantics:

```
(type a)
(block A
  (call B.m1(a)))
(block B
  (macro m1((type x))
    (type a)
    (allow a x (file (read)))))
```

One would expect the parameter `x` of the macro `B.m1` to be bound to the type `a` in the global namespace, thereby allowing `A.a` to read files of type `.a`. Instead, `x` is bound to `A.a`, and the resulting permission for `A.a` is to read files of type `A.a`.

As a second example, consider the following configuration:

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

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 to which name the parameter `x` is bound when the `allow` rule is copied in `B`. If the macro call is resolved before inheritance, then `x` is bound to `.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 behaviour, but the reference guide is unclear about the choice.

a) Ambiguities in CIL: We found cases that are counterintuitive, but nevertheless are represented by our semantics correctly, i.e. in accordance with the actual behaviour of the CIL compiler. For example, the following

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

seems impossible to resolve, because the type `a` defined inside `m` is passed to `m` itself as a parameter. However, this is not deemed to be erroneous according to the compiler's behaviour. Namely, the type `a` is copied from the macro `m` to the block `A`

and then passed as parameter to `m` itself. In a similar puzzling way, if another type named `a` is defined, e.g., in the global environment, it is shadowed by the type copied from the macro.

We also found cases that are meaningless, but are not detected as such by the compiler. In particular this is when typeattributes are recursively defined in a vacuous manner. Consider for example the following configuration:

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

The typeattribute `b` should contain all the elements that are not in itself, which is a contradiction. This error is not detected by the compiler, and a kernel policy is produced whose behaviour cannot be predicted using what we know about the semantics. In fact, 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 misconfigurations may arise silently in complex code where typeattributes are set using macros in different places in the code. Indeed, we found such cases in the openWRT configuration that we used for assessing our tool. Our tool warns the administrator about such misconfigurations and approximates the configuration behaviour by pruning the recursion tree to remove circularity. This misbehaviour from the compiler deserves further investigation.

b) Formal semantics of CIL: To clarify the behaviour of CIL configurations, and to formally support IFCIL and its verification mechanism, 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-word CIL configurations in Section VI-B), and maps each system type to its set of permissions.

In this section, we provide a high-level overview of our CIL semantics. Its detailed formalization is given in Appendix A.

Our semantics benefits from a normal form for configurations. Roughly, we resolve inheritance and macro calls and fully qualify all names. We compute this normal form using the following rewriting pipeline. This pipeline consists of six phases, where each phase repeatedly applies a set of rewrite transformations until the fixed point is reached.

1) The block names in `blockinherit` rules are resolved locally, if possible, or globally otherwise.
2) `blockinherit` rules are replaced by the content of the blocks they refer to.
3) The names of macros in `call` rules are resolved locally, if possible, or globally otherwise.
4) The declarations of types and typeattributes are copied from the body of the macros in the calling blocks.
5) Macro calls are resolved: the type names in the parameters of `call` rules are resolved locally, if possible, or globally otherwise; then the `allow` rules are copied from the macros in the calling blocks. While copying, the non-local names in the `allow` rules are resolved in the block containing the macro definition, if possible; otherwise the resolution is delegated to further application of (5), until no longer possible, and then to (6);
6) The names in `allow` and `typeattributeset` rules in blocks are resolved locally, if possible, or globally if not.

The configuration in the second example is transformed by the first four phases into the left configuration below, where the `(blockinherit A)` first becomes `(blockinherit .A)` and then is resolved as `(call m(a))`; the macro name in the two occurences of `(call m(a))` are both resolved to `.m`; finally the type definition `(type b)` is copied from the macro to blocks `A` and `B`. Phase (5) copies the `allow` rule instantiating the parameter `x` to the names `.a` in `A` and `.B.a` in `B`. Finally, the two occurences of `b` are resolved to `.A.b` and `.B.b`. Note that this representation is that of the binary representation, where names are always fully qualified. The resulting configuration is on the right below.

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

Given a configuration in normal form, our semantic function represents it as a directed labelled graph $G = (N, ta, A)$. The nodes N model the types and the typeattributes (with global names), and the function $ta\colon N \to 2^N$ 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 also the 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 formal definition of the semantic function is straightforward.

For example, the configuration above is associated with the following graph, where ta maps a node into the singleton set containing itself and we omit $\{\}$ for singleton sets on the arcs.

c) Adequacy of the formalization: We define the CIL formal semantics to reflect both the implicit semantics 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 asked CIL developers about the intended behavior [17], [18]. Some cases have been recognized as compiler bugs and the developers will fix them, whereas they will update the documentation in other cases [19].

We identified name resolution as the most involved part of CIL's semantics, especially when name resolution interacts with inheritance or macros.

As an example of a mismatch between the compiler and the reference manual, consider the following configuration:

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

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`. The CIL developers agreed that this is a bug of the compiler [19].

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. The beginning of this section presented several configurations with this kind of problem. Since the manual specifies no evaluation order and even 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 comprehensive testing, discriminating between different orders. Our tests indicate that macro calls are handled after block inheritance (i.e., phases (1) and (2) are executed before phases (3) to (6)). We discovered that no order works for the resolution of different occurrences of block inheritance, and the same applies with different occurrences of macro calls. This is because different occurrences of the same construct are resolved in an interleaved manner. In other words, this resolution consists of a number of steps that are executed in the given order for all the occurrences. For example, all the occurrences of block inheritance must complete phase (1) before any of them starts phase (2). Note that our semantics may seem counterintuitive in some corner cases like those mentioned in paragraph III-a, but it is in agreement with the developers' intent.

IV. The policy language IFCIL

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

A. IFL

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

a) 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 traversed nodes in the graph, and the operations allowed on them. This is done by defining a flow *kind* P using the following grammar.

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

In this grammar, `n` and $\texttt{n}'$ are the starting and the ending nodes in a path of length 1 for `[o]>`, and of length 1 or longer for `+[o]>`. Nodes may also be given using the wildcard $\star$ standing for any node representing a type. The non-empty set $o \subseteq O$ contains a subset of the applicable operations, and it is 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., $\texttt{n [write,read]> n}'$ means that information flows from `n` to $\texttt{n}'$ when `n` writes on $\texttt{n}'$ or $\texttt{n}'$ reads from `n` (the operations in square brakets are the only applicable ones in this step). A direct information flow is represented as a single step $\texttt{n > n}'$, whereas an indirect information flow is represented by multiple steps $\texttt{n +> n}'$. A kind can also mention intermediate steps, e.g., $\texttt{n > } \star \texttt{ > n}'' \texttt{ +> n}'$ specifies that information flows in two steps (through an unspecified node) from `n` to $\texttt{n}''$ and then in multiple steps to $\texttt{n}'$.

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 $\varnothing \neq o' \subseteq o$ 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 $\varnothing \neq o'' \subseteq o$ 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 given by the following grammar

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

for assertions about the information flow diagram I and flow kinds. In particular, the first type of requirement, P, is *path existence*, which stipulates the existence in I of a path π of kind P. The second, $\tilde{}P$, specifies *path prohibition* and requires that there are no paths in I of kind P. The third is *path constraint* and requires that every path π of kind P in I is also of kind P'.

Figure 1 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 2. 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

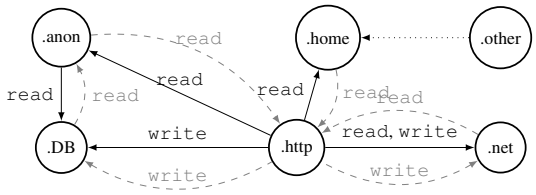

Figure 1. A simple configuration (black solid arcs) and its information flow diagram (gray dashed arcs); the dotted arc represents inclusion of the target in the typeattribute of the source.

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.

b) Formal 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 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*

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

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

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

The second part of the definition has four cases. The first case considers a path in the information flow diagram made 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 $+[o']>$ to $[o']>$. The third case simply iterates the checks along a path longer than one. In the final case, we split a path into 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 $(\mathtt{n}, o, \mathtt{n}') \rhd_I *\ {}_{[o']>}\ \mathtt{n}'$ iff $o \cap o' \neq \varnothing$ holds because the kind $*\ {}_{[o']>}\ \mathtt{n}'$ says that information flows from any node to $\mathtt{n}'$.

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

Definition 2 (validity of a configuration)**.** *Let I be an information flow diagram, and let $\mathcal{R}$ be a requirement of a given configuration. We define I to be valid w.r.t. $\mathcal{R}$, in symbols $I \models \mathcal{R}$, by cases on the syntax of $\mathcal{R}$ as follows:*

$$\begin{aligned}
I \models P \ &\textit{iff}\ \exists \pi \textit{ in } I \textit{ such that } \pi \rhd_I P \\
I \models \tilde{}P \ &\textit{iff}\ \neg(I \models P) \\
I \models P_1 : P_2 \ &\textit{iff}\ \forall \pi \textit{ in } I \textit{ if } \pi \rhd_I P_1 \textit{ then } \pi \rhd_I P_2
\end{aligned}$$

It is immediate to verify that the requirements on the configuration in Figure 1 are indeed satisfied.

c) 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 nontransitive properties, like intransitive noninterference. For example $\mathtt{n} +> \mathtt{n}' : \mathtt{n} +> \mathtt{n}'' +> \mathtt{n}'$ 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 V-A shows. However, IFL allows an additional quantification over paths, which can appear only at top level and is not expressible in LTL.

B. IFCIL

We introduce the language IFCIL, obtained by integrating IFL into CIL. More precisely, we add the following two constructs to augment CIL with comments that specify IFL requirements.

1) *Information flow requirement definitions*, which may occur in blocks and macros. We use them to specify IFL requirements with labels, which must be satisfied by the allow rules of the configuration where they occur. Requirements are copied when calling a macro or inheriting a block, and are managed coherently with the other rules, e.g., concerning name resolution.
2) *Refinement of requirements*, which may occur within `call` and `blockinherit` instructions. Refinements strengthen requirements by further elaborating constraints in the inheriting or caller block.

To ensure backward compatibility, requirement definitions and refinements are enclosed between `;IFL;` thereby taking the form of CIL comments.

The example in Figure 2 illustrates both constructs. For example, the second line in Figure 2 contains a functional requirement labeled with `(F1)` that requires the existence of a

```
(macro in_out((type inp) (type out))
  ;IFL; (F1) inp +> out ;IFL;
  ;IFL; (F2) out +> inp ;IFL;)
(macro anonymize((type x) (type y))
  (type anon)
  (allow anon x (file (read)))
  ;IFL; (S1) x +> y : x > anon +> y ;IFL;)

(typeattribute other)
(typeattributeset other
  (not (or DB (or http (or anon net)))))

(type DB)
(type http)
(type home)
(type net)

(call in_out(net http))

(call in_out(net DB)
  ;IFL; (F1R:F1) * +> http +> * ;IFL;
  ;IFL; (F2R:F2) * +> http +> * ;IFL;)

(call anonymize(DB net)
  ;IFL;(S1R:S1) DB+>net : DB[read]>anon+>net ;IFL;)

(allow http anon (file (read)))
(allow http DB (file (write)))
(allow http other (file (read)))
(allow http net (file (read write)))
;IFL; (S2) ~ DB +> other ;IFL;
```

Figure 2. Example of CIL configuration with IFL annotations.

direct or indirect information flow from the node `inp` to `out`. Nodes in the IFL requirements are types, and are resolved as any other CIL name, e.g., the parameter `out` is bound to `net` in the first call of the macro `in_out`; in contrast, the requirements `F1` and `F2` are simply copied.

In the second call, the administrator duplicates the requirements and refines them with further constraints about how information must flow, since the intermediate node `http` is inserted in the requirements. Note that the new labels refer to those of the original requirements. The new requirements impose that a flow must exist from `net` (instantiating the parameter `inp`) to `DB` (instantiating `out`) and vice-versa, both passing through `http`. Note that since the wildcard is used there is no constraint on the actual parameters. The refined requirement `(F1R:F1)` results then in the path constraint `net +> http : http +> DB`, while the refined requirement `(F2R:F2)` is `DB +> http : http +> net`.

Similarly, in the call to `anonymize`, the requirement `(S1)` is refined by specifying that the operation in the single step is `read`. Of course, the same happens when inheriting a block. Finally, the requirement `(S2)` states that information cannot flow from `DB` to `other`.

We now introduce the most important details of the formalization of IFCIL; its complete definition is in the Appendix B. We first discuss the notion of refinement of IFL requirements. Intuitively, a refinement of a requirement $\mathcal{R}$ allows a subset of the information flow paths allowed by $\mathcal{R}$. This is formalized by a preorder $\preceq$, saying that $\mathcal{R}' \preceq \mathcal{R}$ if $\mathcal{R}'$ refines $\mathcal{R}$; the precise definition of $\preceq$ is given in Appendix B.

We prove the following theorem, stating that the validity of configurations is preserved by refinement.

Theorem 1 (Refinement)**.** *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*

$$I \vDash \mathcal{R}' \Rightarrow I \vDash \mathcal{R}.$$

In defining the semantics of IFCIL, we use the meet of two requirements $\mathcal{R}' \sqcap \mathcal{R}$ on the set of requirements preordered with $\preceq$, i.e., the largest requirement w.r.t. $\preceq$ that is smaller than both $\mathcal{R}'$ and $\mathcal{R}$. To see why, consider the following requirements taken from the example above.

```
;IFL; (F1) inp +> out ;IFL;
;IFL; (F1R:F1) * +> http +> * ;IFL;
```

These requirements are incomparable with respect to $\preceq$. To see this, take I and I' with nodes in {`inp, out, http, a`} such that I has a single arc (`inp, out`), and I' only has the two arcs (`a, http`) and (`http, a`). If they were comparable, Theorem 1 would be falsified because $I \vDash$ `inp +> out` but $I \nvDash$ `* +> http +> *`, and similarly for I' replacing I. Although incomparable, `F1` and `F1R:F1` are clearly related. Namely there exists the meet of the two `F1` $\sqcap$ `F1R:F1` = `inp +> http +> out`. This meet has more details than, and refines both, `F1` and `F1R:F1`, because it requests the presence of an information flow from the node `inp` to `out`, via `http`.

We are now ready to define the semantics of IFCIL. We first normalize configurations by applying the six transformation phases described in Section III, taking meets whenever needed.

The semantics of a configuration consists of a graph G and a set of requirements $\mathbb{R}$ representing the semantics of a CIL configuration and the IFL annotations. It is defined as:

Definition 3 (IFCIL semantics)**.** *Given a (normalized) IFCIL configuration Σ, its semantics is the pair $(G, \mathbb{R})$, where G is the CIL semantics of Σ and $\mathbb{R}$ is the set of IFL requirements occurring in Σ.*

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

Definition 4 (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 $I \vDash \mathbb{R}$, iff $I \vDash \mathcal{R}$ for all $\mathcal{R} \in \mathbb{R}$.*

V. 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 [20] and an IFL requirement as an LTL formula.

A. Encoding in temporal logic

A Kripke transition system (KTS) over a set AP of atomic propositions is $K = (S, Act, \rightarrow, L)$, where S is a set of states, Act is a set of actions, $\rightarrow \subseteq S \times Act \times S$ is a transition relation,

and $L: S \to 2^{AP}$ is a labeling function mapping nodes to a set of proposition that hold at that node. Paths of K are 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 (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 [20], 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 $w \vDash_l \phi$ if the path w of K satifies the LTL formula ϕ; the formal definition is standard and can be found in [20]. 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, disjuction, 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 IV-A in the following equivalent form (recall that the starting node of P in the last two cases is $\mathtt{n}'$).

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

Definition 6 (Encoding of flow kinds)**.** *The encoding of flow kinds is defined as follows.*

$$\llbracket \mathtt{n\ [o]>\ n'} \rrbracket = n \wedge \bigvee_{op \in o} (op) \wedge X(n' \wedge \neg X(true))$$

$$\llbracket \mathtt{n\ +[o]>\ n'} \rrbracket = n \wedge \bigvee_{op \in o} (op) \wedge X(\bigvee_{op \in o} (op)\ U\ (n' \wedge \neg X(true)))$$

$$\llbracket (\mathtt{n\ [o]>\ n'})P \rrbracket = n \wedge \bigvee_{op \in o} (op) \wedge X(\llbracket P \rrbracket)$$

$$\llbracket (\mathtt{n\ +[o]>\ n'})P \rrbracket = n \wedge \bigvee_{op \in o} (op) \wedge X(\bigvee_{op \in o} (op)\ U\ \llbracket P \rrbracket)$$

Note that we use $\neg X(true)$, i.e., the path is complete as there is no next step, to represent the fact that IFL semantics is defined on finite paths.

LTL's semantics can be lifted to a KTS K in quite different manners, with the best for modeling IFL being as follows: $K \vDash_l \phi$ iff $\forall w \in W. w \vDash_l \phi$, 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 7 (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:*

$$K \vdash P \text{ iff } K \nvDash_l \neg\llbracket P \rrbracket$$
$$K \vdash \tilde{}P \text{ iff } K \vDash_l \neg\llbracket P \rrbracket$$
$$K \vdash P : P' \text{ iff } K \vDash_l \neg\llbracket P \rrbracket \vee \llbracket P' \rrbracket$$

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

Note that the three clauses above mimic the analogous clauses in the Definition 2. The first clause says that at least one path satisfies $\llbracket P \rrbracket$; conversely, the second clause says that no path satisfies $\llbracket P \rrbracket$. The third clause simply contains the boolean definition of classical implication.

This correspondence supports the correctness of our verification technique, which is expressed by the following theorem stating that the notions of validity and satisfaction of configurations coincide:

Theorem 2 (Correctness and Completeness)**.** *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*

$$K \vdash \mathbb{R} \quad \textit{if and only if} \quad I \vDash \mathbb{R}.$$

B. Model checking IFCIL

Theorem 2 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 $K \vDash_l \phi$.

In this work, we resort to the classical model checker NuSMV that targets only infinite paths, as usual, whereas Definition 7 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 2 still holds (see Corollary 1 in Appendix B). This extension enables us to verify the correctness of IFCIL configurations with NuSMV.

The worst case complexity of LTL model checking is unfortunately $2^{\mathcal{O}(|\phi|)}\mathcal{O}(|S| + |E'|)$ [20], 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 with respect to the number of names in the requirement. Thus, the complexity of verifying an IFCIL configuration is $\sum_{\phi \in \mathbb{R}} 2^{\mathcal{O}(|\phi|)}\mathcal{O}(|S|)^2 \times \mathcal{O}(|O|)$. Since $|\phi|$ is usually small and does not increase with the configuration size, the complexity grows linearly with respect to

the number of requirements and operations, and quadratically with respect to 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.

VI. 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 a prototype, and not optimized, we were nevertheless able to successfully apply it to large, complex, real-world policies.

A. 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 III. Since other constructs, like those concerning roles, do not affect requirement satisfiability, the tool just ignores them.

For example, the following

```
(type DB)
(type http)
(type home)
(type net)
(type anon)

(typeattribute other)
(typeattributeset .other
  (not (or .DB (or .http (or .anon .net)))))

(allow .anon .DB (file (read)))
(allow .http .anon (file (read)))
(allow .http .DB (file (write)))
(allow .http .other (file (read)))
(allow .http .net (file (read write)))

;IFL; (F1) .net +> .http ;IFL;
;IFL; (F2) .http +> .net ;IFL;
;IFL; (F1R) .net +> .http +> .DB ;IFL;
;IFL; (F2R) .DB +> .http +> .net ;IFL;
;IFL; (S1R) .DB+>.net: .DB[read]>.anon+>.net ;IFL;)
;IFL; (S2) ~ .DB +> .other ;IFL;
```

is the normalization of the configuration in Figure 2. Its IFCIL semantics is the pair $(G, \mathbb{R} = \{$`F1, F2, F1R, F2R, S1R, S2`$\})$, where G is the CIL semantics in Figure 1. It is trivial to derive the KTS K associated with G. To verify the satisfaction of the requirements, we check $K \vdash \mathcal{R} \in \mathbb{R}$. We only show the case $\mathcal{R}$ = `S1R`, i.e., $K \models_l \neg(\!|$`.DB +> .net`$|\!) \vee (\!|$`.DB [read]> .anon +> .net`$|\!)$ where:

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

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

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).

```
MODULE main

DEFINE
  other := (!((type=DB | (type=http |
      (type=anon | type=net))))) & !(type=sink);
VAR
  type : { sink, DB, anon, home, http, net };

IVAR
  operation : { read, write };

TRANS
  (type=DB ->
        ((operation=read & next(type=anon)) |
          next(type=sink))) &
  (type=anon ->
        ((operation=read & next(type=http)) |
          next(type=sink))) &
  (type=home ->
        (next(type=sink))) &
  (type=http ->
        ((operation=write & next(type=DB)) |
          (operation=write & next(type=net)) |
            next(type=sink))) &
  (type=net ->
        ((operation=read & next(type=http)) |
          next(type=sink))) &
  (type=sink -> next(type=sink))

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

We briefly comment on the encoding to generate the input file:

- The state variable `type` 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 `operation` 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 $(\!|\_|\!)$.

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

B. Validation

We experimentally assessed our tool on three real-world CIL policies. The first policy [2] is used in the OpenWrt project,

a version of the Linux operating system targeting embedded devices, like network appliances [21]. The second and the third are SELinux example policies, namely *cilbase* [9] and *dspp5* [10], which serve as templates for creating personalized configurations. The analyzed policies have more than ten thousands lines of code, and make extensive use of all CIL's advanced features, in particular macros and blocks.

To illustrate IFL's expressivity, we formalize various properties that are often considered in the literature, as well as domain-specific policies that we designed. Expressing them in IFL is easy and the resulting requirements are short, direct, and natural. Moreover, we assess the scalability of IFCILverif on real-world examples and show that it scales well to large configurations, checking their requirements in a few seconds.

a) Properties: We first consider the property inspired by Jeager et al. [22], who investigated the *trusted computing base* (TCB) of an SELinux configuration and checked from which types information flows to the TCB, identifying those that do not compromise security. Using IFCIL, the administrator can 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 states that the flow from `a` to `z` must pass through a list of intermediate entities `b`,`c`,... [23], also called *assured pipeline* [24]. It suffices to define requirements of the form `a +> z : * +> b +> c +> ... +> *`.

We express the *wrapping of untrustworthy programs* of [24], by defining requirements stating that all the information flows from (or to) a given type `untrustworthy` must pass through a `verifier` type as first step, i.e., `untrustworthy > * : * > verifier`.

Finally, we propose the additional *augment-only* property that only allows elements of type `a` to increase (`append`) the information on the targets with type `b` without overwriting or removing any. This property is expressed as `a > b : a [append]> b` and `a +> > b : a +> [append]> b`.

b) Experimental results: The results of our analyses on the three configurations are summarized in Table I. For each row, the table reports the kind of property, the number of requirements, and the total time for verifying them (NuSMV input file generation plus LTL model checking). The tool took approximately two minutes to check the entire OpenWRT configuration, and less then three seconds for the other two policies. The 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 the detected violations have on security. This however requires reverse engineering to better understand the security goals of the analyzed policies.

Table I
PERFORMANCE ANALYSIS ON TREE REAL-WORLD CONFIGURATIONS

Property	Requirements	Verification Time
openWRT (45702 lines, 590 types)		
TCB	1	3.50 sec
assured pipeline	3	8,39 sec
wrap untrustworthy	10	12.3 sec
augment only	2	7,86 sec
total	16	29,6 sec
cilbase (11989 lines, 293 types)		
TCB	1	0,063 sec
assured pipeline	4	0,130 sec
wrap untrustworthy	6	0,149 sec
augment only	2	0,116 sec
total	13	0,258 sec
dspp5 (14782 lines, 149 types)		
TCB	1	0,149 sec
assured pipeline	4	0,199 sec
wrap untrustworthy	8	0,299 sec
total	13	0,447 sec

VII. RELATED WORK

Checking the security of the information flow is a problem that has been widely studied since the seminal work by Denning [25], and Goguen and Meseguer [26]. Below, we discuss proposals that target information flow in SELinux policies. 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 [27].

a) 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. [28] 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. [29] 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 [30]; 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 direct information flows caused by operations different from those in `neverallow` and of intransitive information flows that pass through a specific path.

Jaeger et. al. [22] 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. [23] 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. [31] propose a logic-programming based approach to analyzing SELinux policies. Their tool transforms a configuration into a Prolog program, thus allowing the administrator to perform deductions on the properties of the configuration 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.

Finally, high-level languages have been proposed for SELinux based on information flows. 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. [6] 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 is backward compatible.

Nakamura et al. [14] 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.

b) Information flow on access control: Bugliesi et al. [32] develop a verification framework supported by a tool for `grsecurity`, a role-based access control system for Unix/Linux [33]. 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. [34] continue this line of research by proposing a security type system for verifying information flow in ARBAC policies. In particular, their type system can address the role reachability problem and offers a compositional technique. Also in these policies are built into the type system and do not cover intransitive properties. In contrast, our policies are written in a language, can be composed, and can express intransitive properties. Since we use model checking, the results of our verification phase cannot, however, be composed.

Guttman and Herzog [35] 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.

c) Adding information flow to real 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 [37] 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 [38] 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 [39] 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 [40] extends Java with information flow policies building on an object-oriented generalisation of Paralocks [41]. 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.

VIII. 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 CIL extension. 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.

a) Discussion: We believe that our extension can help with 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 [42]. 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 VI-B, our tool deals with all the type enforcement constructs used in real-word 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 [30].

b) 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-word 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, and extending it to fully support requirement refinement. 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 the case 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 [16].

ACKNOWLEDGEMENTS

We thank the anonymous referees and the shepherd for their insightful comments and detailed suggestions that helped to greatly improve our presentation. L. Ceragioli, P. Degano and L. Galletta have been partially supported by the MIUR project PRIN 2017FTXR7S IT MATTERS.

References

[1] T. Yokoyama, M. Hanaoka, M. Shimamura, K. Kono, and T. Shinagawa, "Reducing security policy size for internet servers in secure operating systems," *IEICE Trans. Inf. Syst.*, vol. 92-D, 11, pp. 2196–2206, 2009.

[2] D. Grift, "openWRT SELinux policy (commit aa59f95 on 3 Dec 2021)," https://git.defensec.nl/?p=selinux-policy.git;a=summary.

[3] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *20th Network and Distributed System Security Symposium*. The Internet Society, 2013.

[4] B. Im, A. Chen, and D. S. Wallach, "An historical analysis of the SEAndroid Policy Evolution," in *Procs. 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 629–640.

[5] stephensmalley, "Add support for a source policy hll," https://github.com/SELinuxProject/selinux/issues/54, 2018.

[6] J. Hurd, M. Carlsson, B. Letner, and P. White, "Lobster: A domain specific language for selinux policies," Galois, Inc., Tech. Rep., 2008.

[7] "Selinux ifcil tool," https://github.com/lceragioli/SELinuxIFCIL.

[8] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *14th CAV*, ser. LNCS, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Springer, 2002, pp. 359–364.

[9] D. Grift, "SELinux example policy cilpolicy (commit 562a8a2 on 8 Sep 2015)," https://web.archive.org/web/20201027211840/https://github.com/doverride/cilpolicy.

[10] ——, "SELinux example policy dspp5 (commit cf4dd60 on 16 Dec 2021)," https://git.defensec.nl/?p=dssp5.git;a=summary.

[11] L. Ceragioli, L. Galletta, P. Degano, and D. Basin, "Ifcil: An information flow configuration language for selinux (extended version)," 2022. [Online]. Available: https://arxiv.org/abs/2205.15915

[12] "Selinux project," https://selinuxproject.org.

[13] "The SELinux Notebook - Kernel Policy Language," https://github.com/SELinuxProject/selinux-notebook/blob/main/src/kernel_policy_language.md#kernel-policy-language.

[14] Y. Nakamura, Y. Sameshima, and T. Yamauchi, "Selinux security policy configuration system with higher level language," *J. Inf. Process.*, vol. 18, pp. 201–212, 2010.

[15] "The SELinux Notebook - CIL Reference Guide (commit 2b3b4ee on 6 Jul 2020)," https://github.com/SELinuxProject/selinux-notebook/blob/main/src/notebook-examples/selinux-policy/cil/CIL_Reference_Guide.pdf.

[16] "sepolicy," https://android.googlesource.com/platform/system/sepolicy/.

[17] L. Ceragioli, "Bug (?) report for secilc and cil semantics: some unexpected behaviours," https://lore.kernel.org/selinux/5ca2e18c-6395-a0af-fdee-b0ac5f1de714@phd.unipi.it/.

[18] ——, "[bug report?] other unexpected behaviours in secilc and cil semantics," https://lore.kernel.org/selinux/86d254dd-fd82-e25c-915b-16615b341457@phd.unipi.it/.

[19] J. Carter, "[patch 1/3] libsepol/cil: Make name resolution in macros work as documented," https://lore.kernel.org/selinux/20210507173744.198858-1-jwcart2@gmail.com/.

[20] M. Müller-Olm, D. A. Schmidt, and B. Steffen, "Model-checking: A tutorial introduction," in *Procs 6th Static Analysis Symposium, LNCS,1694*, A. Cortesi and G. Filé, Eds. Springer, 1999, pp. 330–354.

[21] https://openwrt.org.

[22] T. Jaeger, R. Sailer, and X. Zhang, "Analyzing integrity protection in the selinux example policy," in *Procs 12th USENIX Security Symposium, Washington, D.C., USA, 2003*. USENIX Association, 2003.

[23] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka, "Verifying information flow goals in security-enhanced linux," *J. Comput. Secur.*, vol. 13, no. 1, pp. 115–134, 2005.

[24] D. Thomsen, "Information flow analysis in security enhanced linux," CERIAS Security Seminar at Purdue University.

[25] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, p. 236243, may 1976.

[26] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–11.

[27] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[28] E. Reshetova, F. Bonazzi, and N. Asokan, "Selint: An seandroid policy analysis tool," in *Procs 3rd Int'l Conf. on Information Systems Security and Privacy*, P. Mori, S. Furnell, and O. Camp, Eds. SciTePress, 2017, pp. 47–58.

[29] B. S. Radhika, N. V. N. Kumar, R. K. Shyamasundar, and P. Vyas, "Consistency analysis and flow secure enforcement of selinux policies," *Comput. Secur.*, vol. 94, p. 101816, 2020.

[30] Google, "Android open source project," https://source.android.com/.

[31] B. Sarna-Starosta and S. D. Stoller, "Policy analysis for security-enhanced linux," in *Procs 2004 Workshop on Issues in the Theory of Security*, 2004, pp. 1–12.

[32] M. Bugliesi, S. Calzavara, R. Focardi, and M. Squarcina, "Gran: Model checking grsecurity RBAC policies," in *25th IEEE Computer Security Foundations Symposium,*, S. Chong, Ed., 2012, pp. 126–138.

[33] B. Spengler, "Increasing performance and granularity in rolebased access control systems," http://grsecurity.net/researchpaper.pdf, 2004.

[34] S. Calzavara, A. Rabitti, and M. Bugliesi, "Compositional typed analysis of ARBAC policies," in *IEEE 28th Computer Security Foundations Symposium*, C. Fournet, M. W. Hicks, and L. Viganò, Eds., 2015, pp. 33–45.

[35] J. D. Guttman and A. L. Herzog, "Rigorous automated network security management," *Int. J. Inf. Sec.*, vol. 4, no. 1-2, pp. 29–48, 2005.

[36] F. Pottier and V. Simonet, "Information flow inference for ml," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, p. 117158, jan 2003.

[37] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, p. 228241.

[38] J. Liu, O. Arden, M. D. George, and A. C. Myers, "Fabric: Building open distributed systems securely by construction," *J. Comput. Secur.*, vol. 25, no. 4-5, pp. 367–426, 2017.

[39] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, "Liquid information flow control," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, aug 2020.

[40] N. Broberg, B. van Delft, and D. Sands, "Paragon - practical programming with information flow control," *J. Comput. Secur.*, vol. 25, no. 4-5, pp. 323–365, 2017.

[41] N. Broberg and D. Sands, "Paralocks: role-based information flow control and beyond," in *Proceedings of the 37th POPL*, M. V. Hermenegildo and J. Palsberg, Eds. ACM, 2010, pp. 431–444.

[42] "The SELinux Notebook - MLS Statements," https://github.com/SELinuxProject/selinux-notebook/blob/main/src/mls_statements.md#mls-statements.

Appendix A
Formalizing CIL

CIL has qualified names, i.e., names prefixed by a path ρ in the nesting of blocks and macros. ρ is a list of elements separated by dots. In CIL, qualified names from the global namespace start with a dot (.). We instead use the distinguished symbol $\#$. Global qualifications σ start from the global namespace $\#$; other qualifications are called relative (e.g., $\#.A.a$ is globally qualified, $A.a$ is a relatively qualified).

The syntax of a CIL configuration is as follows, where $[F]$ represents lists of F entities, and CIL is the starting symbol.

$$
\begin{aligned}
CIL &::= [rule] \\
rule &::= declaration \mid command \\
declaration &::= (block\ n\ CIL) \mid (typeattribute\ n) \\
&\quad \mid (type\ n) \mid (macro\ n([x])(CIL)) \\
command &::= (allow\ a\ a\ (class\ (perms))) \\
&\quad \mid (typeattributeset\ a\ (expr)) \\
&\quad \mid (call\ m([a])) \mid (blockinherit\ B)
\end{aligned}
$$

Here $n, n', \ldots$ are unqualified names (of types, typeattributes, macros or blocks); $x, x', \ldots$ are formal parameter names; $a, a', \ldots$ are types and typeattributes (possibly qualified); $m, m', \ldots$ are macro (possibly qualified) names; $B, B', \ldots$ are block (possibly qualified) names. Furthermore, we will use $g, g', \ldots$ for possibly qualified names of macros or blocks, and

$$\text{(N-1)}\ \frac{\overline{eval}_{\sigma;\#}(B) = B'}{(\sigma, inherit\ B) \to (\sigma, inherit\ B')} \qquad \text{(N-2)}\ \frac{(\sigma, inherit\ B) \in \Gamma \quad (B.\rho, r) \in \Gamma}{add(\sigma.\rho, r)} \qquad \text{(N-3)}\ \frac{\overline{eval}_{\sigma;\#}(m) = m'}{(\sigma, call\ m([a])) \to (\sigma, call\ m'([a]))}$$

$$\text{(N-4)}\ \frac{(\sigma, call\ m([a])) \in \Gamma \quad (m, d) \in \Gamma}{add(\sigma, d)} \qquad \text{(N-5a)}\ \frac{a\ occurs\ in\ c \quad \overline{eval}_m(a) = a' \neq \bot \quad eval_m(a) = \bot}{(m, c) \to ((m, c\{a'/a\})} \qquad \text{(N-5b)}\ \frac{\begin{array}{c}(\sigma, call\ \sigma'.n([a])) \in \Gamma \\ (\sigma'.n, c) \in \Gamma \\ (\sigma', macro\ m([x])) \in \Gamma \\ (\neg\exists m', a' : (\sigma'.n, call\ m'[a']) \in \Gamma)\end{array}}{add(\sigma, c\{[a]/[x]\})}$$

$$\text{(N-5c)}\ \frac{\begin{array}{c}(\sigma, call\ m([a])) \in \Gamma \\ (\neg\exists m', [a'] : (m, call\ m'[a']) \in \Gamma)\end{array}}{remove(\sigma, call\ m([a]))} \qquad \text{(N-6)}\ \frac{a\ occurs\ in\ c \quad \overline{eval}_{B_\#;\#}(a) = a'}{(B_\#, c) \to (B_\#, c\{a'/a\})}$$

Figure 3. CIL normalization rules.

$$\text{(S-N1)}\ \frac{(B_\#, type\ n) \in \Gamma}{B_\#.n \in N} \qquad \text{(S-N2)}\ \frac{(B_\#, typeattribute\ n) \in \Gamma}{B_\#.n \in N} \qquad \text{(S-ta1)}\ \frac{(B_\#, type\ n) \in \Gamma}{(B_\#.n, B_\#.n) \in ta}$$

$$\text{(S-ta2)}\ \frac{\begin{array}{c}(B_\#, typeattributeset\ a\ (expr))) \in \Gamma \\ (B'_\#, type\ n) \in \Gamma \\ [\![expr]\!]_\Gamma(B'_\#.t)\end{array}}{(a, B'_\#.t) \in ta} \qquad \text{(S-A1)}\ \frac{(B_\#, allow\ t\ t'\ (class\ (perms))) \in \Gamma}{(t, perms, t') \in A} \qquad \text{(S-A2)}\ \frac{\begin{array}{c}(t1, perms, t2) \in A \\ t1' \in ta(t1) \\ t2' \in ta(t2)\end{array}}{(t1', perms, t2') \in A}$$

Figure 4. CIL semantics.

$p, p', \ldots$ for possibly qualified names in general. Finally, we use $B_\#, B'_\#, \ldots$ to refer to either $\#$ or a block name B.

We abstractly represent a CIL configuration as a set of pairs (σ, r), where the rule r occurs in the namespace σ.

Assume as given a set of CIL rules Γ. We define the following function $eval^k_\sigma(p)$ to resolve names, where $p = \rho.n$ (the qualification ρ is possibly empty, and n is the unqualified name) occurring in the globally qualified block or macro σ, and where $k \in \{\texttt{type}, \texttt{typeattribute}, \texttt{block}, \texttt{macro}\}$ indicates that we are resolving a type, a typeattribute, etc. This function returns a fully qualified name for p, or $\bot$ is resolution is not possible in σ.

$$eval^k_\sigma(\rho.n) = \begin{cases} \rho.n & \text{if } \rho = \#.\rho' \\ \sigma.\rho.n & \text{if } \rho = \#.\rho' \ \wedge (\sigma.\rho, k\ n) \in \Gamma \\ \bot & \text{otherwise} \end{cases}$$

Since typeattributes are treated as types in CIL, we abuse notation and simply write $eval^{type}_\sigma(a)$ for the function defined as $eval^{type}_\sigma(a)$ if its result is different from $\bot$, and as $eval^{typeattribute}_\sigma(a)$ otherwise. Moreover, we omit k, assuming that the correct parameter is used. In CIL, a name that cannot be resolved in the namespace in which it occurs is often resolved in its parent namespace (recursively). We formalize this as follows

$$\overline{eval}^k_\sigma(p) = \begin{cases} eval^k_\sigma(p) & \text{if } eval^k_\sigma(p) \neq \bot \\ \overline{eval}^k_{\sigma'}(p) & \text{if } eval^k_\sigma(p) = \bot \ \wedge \\ & \sigma = \sigma'.n \ \texttt{with}\ \sigma' \neq \# \\ \bot & \text{otherwise} \end{cases}$$

Moreover, it is common that a name is resolved in the global namespace if the resolution in the block or macro in which the name is used fails. We will write $eval_{\sigma;\sigma'}(p)$ for a function that, evaluates p in σ unless the result is $\bot$, and evaluates it in σ' otherwise.

The CIL normalization pipeline consists of the six rewriting rules in Figure 3, with applicability conditions in the upper part and an action in the lower one. We denote rules by $r, r', \ldots$; declarations by $d, d', \ldots$; and commands by $c, c', \ldots$. The conditions predicate on the configuration at hand, and the actions either prescribe: (i) to rewrite a rule in Γ, as in $(\sigma, r) \to (\sigma', r')$; (ii) to add a rule in Γ, as in $add\ (\sigma, r)$; and (iii) to remove a rule from Γ, as in $remove(\sigma, r)$. Each phase is iterated until a fixpoint is reached, with the only exception being the fifth phase. The fifth phase is a sub-pipeline where rule (N-5a) is applied first, then rule (N-5b), and finally (N-5c). In the fifth phase, each rule and the whole pipeline are applied until a fixpoint is reached.

The rules for CIL's semantics are given in Figure 4, which yield the graph $G = (N, ta, A)$. Since attribute expressions $expr$ are boolean functions on types and typeattributes, we assume a denotational semantics $[\![expr]\!] : N \to \{\text{true}, \text{false}\}$.

APPENDIX B
FORMALIZING IFCIL

A. IFL

IFL requirement refinement is defined by the $\preceq$ operator in Figure 5, where reflexivity and transitivity rules are implicitly assumed for every defined relation, and where arrows > or

$$\text{(node)}\ \frac{-}{n \preceq_n *} \qquad \text{(op)}\ \frac{o \subseteq o'}{o \preceq_o o'} \qquad \text{(arrow)}\ \frac{-}{> \ \preceq_a \ + >} \qquad \text{(o-arrow)}\ \frac{o \preceq_o o' \quad w \preceq_a w'}{(w, o) \preceq_{oa} (w', o')} \qquad \text{(comp)}\ \frac{P_1 \preceq_P P_1' \quad P_2 \preceq_P P_2'}{P_1 P_2 \preceq_P P_1' P_2'}$$

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

$$\text{(P-3)}\ \frac{n_1 \preceq_n n_1' \quad n_2 \preceq_n n_2' \quad o_1 \preceq_o o' \quad o_2 \preceq_o o'}{n_1\ +[o_1] >\ *\ +[o_2] > n_2 \preceq_P n_1'\ +[o'] >\ n_2'} \qquad \text{(P-4)}\ \frac{n_1 \preceq_n n_1' \quad n_2 \preceq_n n_2' \quad o_1 \preceq_o o_1' \quad o_2 \preceq_o o_2' \quad o_2 \preceq_o o_1'}{n_1\ [o_1] >\ *\ +[o_2] > n_2 \preceq_P n_1'\ +[o_1'] >\ *\ [o_2'] >\ n_2'}$$

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

Figure 5. Definition of IFL requirement refinement and auxiliary relations.

$+ >$ are sometimes represented by w, and an arrow w labeled with a set of operations o is represented as a pair (w, o) (e.g., $+[\{read\}] >$ is represented as $(+ >, \{read\})$).

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

Sketch of the proof. Rule induction suffices: for each rule in Figure 5 we prove the property for the conclusion assuming that the premises enjoy the property. □

Theorem 1 (Refinement)**.** *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*

$$I \vDash \mathcal{R}' \Rightarrow I \vDash \mathcal{R}.$$

Sketch of the proof. Rule induction suffices: for rules ($\mathcal{R}$-1), ($\mathcal{R}$-2), and ($\mathcal{R}$-3) of Figure 5 we assume that the lemma holds for the premises, and then we prove that it also holds for the conclusions. □

The (not always defined) *greatest lower bound*, and *least upper bound* of a pair of requirements $\mathcal{R}$ and $\mathcal{R}'$ are represented as $\mathcal{R} \sqcap \mathcal{R}'$, and $\mathcal{R} \sqcup \mathcal{R}'$ respectively.

B. Syntax and semantics of IFCIL

The grammar for IFCIL is obtained by updating the rules of *command* for *call* and *blockinherit* as follows.

$$\begin{aligned} \mathit{command} ::= &\ (\mathit{call}\ m([a])\ [\mathit{IFLrefinement}]) \\ \mid &\ (\mathit{blockinherit}\ B\ [\mathit{IFLrefinement}]) \end{aligned}$$

Moreover, the following rules are added to the grammar.

$$\begin{aligned} \mathit{command} &::= \mathit{IFLrequirement} \\ \mathit{IFLrequirement} &::= \ ;\texttt{IFL;}\ (\mathit{label})\ \mathcal{R}\ ;\texttt{IFL;} \\ \mathit{IFLrefinement} &::= \ ;\texttt{IFL;}\ (\mathit{label} : \mathit{label})\ \mathcal{R}\ ;\texttt{IFL;} \end{aligned}$$

In the normalization pipeline, the rules are updated as shown in Figure 6, where $\overline{r}$ and $\overline{c}$ are rules and commands that are not IFL requirements. The normalization procedure is the same as CIL, where rules (N-i), (N-i'), and (N-i") are applied together during phase i ($1 \le i \le 6$).

The semantics $(G, \mathbb{R})$ of a IFCIL configuration Σ is obtained by applying the rules in Figure 4 for G, and the following one for $\mathbb{R}$.

$$(\text{S-}\mathbb{R})\ \frac{(B_{\#}, ;\texttt{IFL;}(l)\ \mathcal{R}\ ;\texttt{IFL;}) \in \Gamma}{\mathcal{R} \in \mathbb{R}}$$

C. Verification

It is convenient to use in the following the grammar of P of Section V, and to redefine the relation $\rhd_I$ of Definition 1 in the following, trivially equivalent way.

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

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

Lemma 2. *Let π be a path in the information diagram I with K its KTS, and let P be an information flow kind. Then $\pi \rhd_I P$ if and only if $w \vDash_l (\!|P|\!)$ for some $w \in (\!|\pi|\!)$.*

Sketch of the proof. The lemma trivially follows by structural induction on P. □

Lemma 3. *Let w be a path in the KTS K of the information diagram I, and let P be an information flow kind. Then $w \vDash_l (\!|P|\!)$ if and only if $\pi_w \rhd_I P$.*

$$\text{(N-1)}\ \frac{\overline{eval}_{\sigma;\#}(B) = B'}{(\sigma, inherit\ B\ R) \to (\sigma, inherit\ B'\ R)} \quad \text{(N-2)}\ \frac{(\sigma, inherit\ B\ R) \in \Gamma \quad (B.\rho, \overline{r}) \in \Gamma}{add(\sigma.\rho, \overline{r})} \quad \text{(N-2')}\ \frac{\begin{array}{c}(\sigma, inherit\ B\ R) \in \Gamma \\ (B.\rho, \texttt{;IFL;}\ (l)\ \mathcal{R}\ \texttt{;IFL;}) \in \Gamma \\ \texttt{;IFL;}(l' : \rho.l)\ \mathcal{R}'\texttt{;IFL;} \in R\end{array}}{add(\sigma.\rho, \texttt{;IFL;}(l')\ \mathcal{R}' \sqcap \mathcal{R}\ \texttt{;IFL;})}$$

$$\text{(N-2'')}\ \frac{\begin{array}{c}(\sigma, inherit\ B\ R) \in \Gamma \\ (B.\rho, \texttt{;IFL;}\ (l)\ \mathcal{R}\ \texttt{;IFL;}) \in \Gamma \\ \neg\exists l', \mathcal{R}' : \texttt{;IFL;}(l' : \rho.l)\ \mathcal{R}'\texttt{;IFL;} \in R\end{array}}{add(\sigma.\rho, \texttt{;IFL;}(l')\ \mathcal{R}\ \texttt{;IFL;})} \quad \text{(N-3)}\ \frac{\overline{eval}_{\sigma;\#}(m) = m'}{(\sigma, call\ m([a])\ R) \to (\sigma, call\ m'([a])\ R)} \quad \text{(N-4)}\ \frac{\begin{array}{c}(\sigma, call\ m([a])\ R) \in \Gamma \\ (m, d) \in \Gamma\end{array}}{add(\sigma, d)}$$

$$\text{(N-5a)}\ \frac{\begin{array}{c}a\ occurs\ in\ c \\ \overline{eval}_m(a) = a' \neq \bot \\ eval_m(a) = \bot\end{array}}{(m, c) \to ((m, c\{a'/a\})} \quad \text{(N-5b)}\ \frac{\begin{array}{c}(\sigma, call\ \sigma'.n([a])\ R) \in \Gamma \\ (\sigma'.n, \overline{c}) \in \Gamma \\ (\sigma', macro\ m([x])) \in \Gamma \\ (\neg\exists m', a', R' : (\sigma'.n, call\ m'[a']\ R') \in \Gamma)\end{array}}{add(\sigma, \overline{c}\{[a]/[x]\})}$$

$$\text{(N-5b')}\ \frac{\begin{array}{c}(\sigma, call\ \sigma'.n([a])\ R) \in \Gamma \\ (\sigma'.n, \texttt{;IFL;}(l)\ \mathcal{R}\ \texttt{;IFL;}) \in \Gamma \\ (\sigma', macro\ m([x])) \in \Gamma \\ (\neg\exists m', a', R' : (\sigma'.n, call\ m'[a']\ R') \in \Gamma) \\ \texttt{;IFL;}(l' : l)\ \mathcal{R}'\texttt{;IFL;} \in R\end{array}}{add(\sigma, \texttt{;IFL;}(l')\ (\mathcal{R} \sqcap \mathcal{R}')\{[a]/[x]\}\ \texttt{;IFL;})} \quad \text{(N-5b'')}\ \frac{\begin{array}{c}(\sigma, call\ \sigma'.n([a])\ R) \in \Gamma \\ (\sigma'.n, \texttt{;IFL;}(l)\ \mathcal{R}\ \texttt{;IFL;}) \in \Gamma \\ (\sigma', macro\ m([x])) \in \Gamma \\ (\neg\exists m', a', R' : (\sigma'.n, call\ m'[a']\ R') \in \Gamma) \\ \neg\exists l', \mathcal{R}' : \texttt{;IFL;}(l' : l)\ \mathcal{R}'\texttt{;IFL;} \in R\end{array}}{add(\sigma, \texttt{;IFL;}(l)\ \mathcal{R}\{[a]/[x]\}\ \texttt{;IFL;})}$$

$$\text{(N-5c)}\ \frac{\begin{array}{c}(\sigma, call\ m([a])\ R) \in \Gamma \\ (\neg\exists m', [a'] : (m, call\ m'[a']\ R) \in \Gamma)\end{array}}{remove(\sigma, call\ m([a])\ R)} \quad \text{(N-6)}\ \frac{\begin{array}{c}a\ occurs\ in\ c \\ \overline{eval}_{B_\#;\#}(a) = a'\end{array}}{(B_\#, c) \to (B_\#, c\{a'/a\})}$$

Figure 6. IFCIL normalization rules.

Proof. Trivially derives from Lemma 2 since $(\!|\pi_w|\!) = \{w\}$. □

Theorem 2 (Correctness and Completeness)**.** *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*

$$K \vdash \mathbb{R} \ \text{ if and only if } \ I \models \mathbb{R}.$$

Sketch of the proof. By cases on $\mathcal{R}$ using Lemmas 2 and 3. □

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 8. *The encoding of flow kinds for K_ι is as follows.*

$$(\!|n\ \texttt{[o]>}\ n'|\!)_\iota = n \wedge \bigvee_{op \in o} (op) \wedge X(n' \wedge X(\iota))$$

$$(\!|n\ \texttt{+[o]>}\ n'|\!)_\iota = n \wedge \bigvee_{op \in o} (op) \wedge X(\bigvee_{op \in o} (op)\ U\ (n' \wedge X(\iota)))$$

$$(\!|(n\ \texttt{[o]>}\ n')P|\!)_\iota = n \wedge \bigvee_{op \in o} (op) \wedge X((\!|P|\!)_\iota)$$

$$(\!|(n\ \texttt{+[o]>}\ n')P|\!)_\iota = n \wedge \bigvee_{op \in o} (op) \wedge X(\bigvee_{op \in o} (op)\ U\ (\!|P|\!)_\iota)$$

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

$$K_\iota \vdash_\iota P \text{ iff } K_\iota \not\models_l \neg(\!|P|\!)_\iota$$
$$K_\iota \vdash_\iota \tilde{}P \text{ iff } K_\iota \models_l \neg(\!|P|\!)_\iota$$
$$K_\iota \vdash_\iota P : P' \text{ iff } K_\iota \models_l \neg(\!|P|\!)_\iota \vee (\!|P'|\!)_\iota$$

Lemma 4. *If $w \models_l (\!|P|\!)$ then $w \in \dot{W}$ and $w\iota^\omega \models_l (\!|P|\!)_\iota$ with $w\iota^\omega \in \bar{W}_\iota$.*

Sketch of the proof. By structural induction on P. □

Lemma 5. *If $w \models_l (\!|P|\!)_\iota$ then there exists a unique $\dot{w} \in \dot{W}$ such that $\dot{w} \models_l (\!|P|\!)$ and $w = \dot{w}\iota^\omega$.*

Sketch of the proof. By structural induction on P. □

Corollary 1. *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*

$$K_\iota \vdash \mathbb{R} \ \text{ if and only if } \ K \vdash \mathbb{R} \ \text{ if and only if } \ I \models \mathbb{R}.$$

Sketch of the proof. It suffices to prove the two implications below, which follow from Lemmas 4 and 5.

$$K \models_l \neg(\!|P|\!) \text{ iff } K_\iota \models_l \neg(\!|P|\!)_\iota \tag{1}$$
$$K \models_l \neg(\!|P|\!) \vee (\!|P'|\!) \text{ iff } K_\iota \models_l \neg(\!|P|\!)_\iota \vee (\!|P'|\!)_\iota \tag{2}$$

□