Università di Pisa

Dipartimento di Informatica
Ph.D. Programme in Computer Science

Ph.D. Thesis

Access Control Policies Across Abstraction Layers

Lorenzo Ceragioli

Supervisors:

Pierpaolo Degano
Letterio Galletta

Referees:

David Basin
Rosario Pugliese

Contents

Abstract

System administrators specify the access control policy they want and implement the relevant configuration for enforcing it. Specifying policies and implementing configurations require users to switch from one level of abstraction to another, often changing language and tools. The gap between these two abstraction layers seems to be a widespread problem, and may cause poor security and low efficiency.

Our thesis aims at proposing solutions, based on formal semantics and logic, for security engineers to interact with access control systems at different abstraction layers for configuring, updating and verifying system behaviour. We consider different contexts: networks, operating systems and collaborative environments. For each of them we formally model the low level of the executable configurations and propose a high level language inspired by the needs of policy designers.

For network security, we propose FWS, a tool that allows the administrator to manage a firewall configuration written in a legacy language (`iptables`, `pf`, `ipfw`) abstracting away from low level details like shadowing, tags and the limitations of packet matching. The idea is to provide the administrator with a declarative description of the behaviour, and with the means for reasoning about the system security. The possibly modified declarative description can be then compiled back to the preferred language. This also offers the possibility for transcompilation, but contrary to the expectation, legacy languages are not equally expressive. We study the expressive power of Unix firewall languages and propose the tool *F2F* to check if a given configuration is expressible in another firewall language.

For system security, we target SELinux configurations. We propose IFL, a domain specific language for defining fine grained information flow requirements (including confidentiality, integrity and non-transitive properties). IFL expresses both functional requirements, i.e., which permissions must be granted to users for performing their authorized tasks, and security requirements, i.e., information flows to forbid because possibly dangerous. We extend CIL, a SELinux policy language for writing structured configurations, with IFL requirements. In this way, the administrator abstracts from the details of the system, expressing desired high level properties of the configuration. A verification procedure is given and implemented in the tool IFCILverif that statically checks that all the requirements are met in a configuration. Moreover, we empirically validate our formal semantics of CIL, and discover some unspecified corner cases and disagreements between the documentation and the compiler.

Finally, we consider collaborative environments, in which the users interact by sharing or exchanging assets for mutual advantage. We designed MuAC, an high level language with which each user can express what they want in return for allowing access to their assets. We address both the case of infinite

or reusable assets, where the asset is still available to the owner after he allows access to it, and the case of finite resources that are consumed when exchanged. The low level is a non-standard logical theory that also gives semantics to MuAC policies, and the evaluation of access requests rely on logical deductions. Finally, we propose a compilation from abstract policies to logical theories, and an algorithm for deciding access requests by changing ownership of the resources or updating an access control matrix, driving the system to behave correctly.

Acknowledgement

The work presented here was only possible thanks to the support of my advisors Prof. Pierpaolo Degano and Dott. Letterio Galletta: my warmest thanks. I would also like to express my gratitude to Prof. Nicola Zannone and Prof. Joshua D. Guttman for their useful suggestions and comments on the first version of this thesis. Finally, I want to thank Prof. David Basin and Prof. Luca Viganò for being wonderful hosts during my visits.

Chapter 1

Introduction

Access control mechanisms are usually the first line of defence in computing systems. They selectively restrict access to resources, distinguishing between legitimate and illegitimate requests. These mechanisms perform two tasks: preventing unauthorized users from gaining access to resources, and enabling authorized users to access resources in a regulated manner.

This thesis will adopt the standard terminology: *subjects* are the users that may want to perform certain actions; *resources* are the possible objects of requests, they are the assets that we have to protect; *operations* are the possible actions that the subjects may perform on some resource; *requests* are communications from a subject to the system in which he asks the permission to perform an operation on a resource. Access control is strictly correlated to authentication. Anyway, we focus only on actual access control, abstracting from the issues of authenticating access requests, for which different technologies are nowadays available.

Providing a configuration is the most common way of describing how the system should evaluate the requests. Some alternative methods exist, like relying on the ability of the device to reason or to perceive the environment [30]. Here we will focus on policy based access control.

Access control requires enforcement mechanisms that act on the target system for preventing undesired behaviour. To reflect the needs and the security concerns of the system, a *configuration* must be produced for these mechanisms. We call *low level* the one that faithfully represents the system and the enforcing mechanisms, where the prescription of the configuration are actually implemented. For example, on a common Unix system, the typical enforcing mechanism receives requests for performing operations on a given file, and knows the owner of the file and the username and group of the requester, and a configuration can only be based on very concrete models.

Above there is the *high level* that is used for abstractly reasoning about the system security. It hides the implementation details making easier to define strategies and desired properties. The high level is used when designing abstract policies, that we call *specifications*, and that encode the rules according

to which access control must be regulated. Configurations and requirements have the same purpose, i.e., defining the correct behaviour, but only the first one can actually be implemented by the enforcement mechanisms of the underlying system, whereas the second one is more adequate for expressing and reasoning about security. Hereafter, we use the term *policy* to refer either to the high or low level description.

Different tasks must be performed, e.g., defining the security properties, verifying their correctness, implementing an executable configuration, testing its performance and evaluating its robustness. Each of these requires different languages and tools, and is better performed at one of the abstraction layers. In the field of access control, the gap between these two abstraction layers seems to be a widespread problem, causing poor security and efficiency.

Consider for example a Linux server with multiple services that shares a common file system. In general, the different services may accede a separate part of the operating system, but some interaction may be needed, either through sockets or shared files. At the high level, the administrator collects the security requirements of the system, that speak about isolation, confidentiality and non-interference between different entities (users and services). The language used for these requirements may be a logical one, that can be analyzed resorting to standard techniques and tools. The access control system of the server may be either SELinux or the usual Unix discretionary access control (or both), and an actual configuration must be produced to enforce the desired behavior. When writing the configuration, the administrator must encode the high level requirements into a low level language, that is either very intricate (like the one of SELinux), or provides no high level features (like Unix Access Control Lists). In both cases, manually coding is time consuming and error prone, and the configuration must be checked for possible misconfigurations.

We propose an approach for enabling security engineers to interact with access control systems at different abstraction layers for configuring, updating and verifying system behaviour. The main idea is to keep both the low and the high level representations, and to maintain them coherent also when they change. A special care is needed with respect to specific elements, like the management of low level details and the limitations of the underlying system.

We apply our approach in three different contexts: network security, system security and collaborative environment. For each of them we formally model both high and low level, and grant coherence with different means.

Structure of the Chapter

In Section 1.1 we state the problem and present our approach in a general setting. In Section 1.2 we briefly describe how we instantiate the two-layers approach on different contexts: firewalls, SELinux configurations, and policies for exchanging access rights in collaborative environments. In Section 1.3 we compare our approach with similar ones in the literature. In Section 1.4 we present the plan of the thesis. Finally, in Section 1.5 we briefly discuss the candidate's publications that share parts of the thesis.

1.1 Two Layers Approach

In this section we introduce the problem that we address, discuss its relevance and which are the requirements for a good solution. Then we present our proposed approach, and how to apply it in different contexts.

1.1.1 The Gap Between Specification and Implementation

The world of Access Control (AC) is large and various, with lots of challenges related to the very specific domains. In Big Data, for example, one of the main problems seems to be a missing common framework among different technologies [41], while in social networks the user and his relationships are central when evaluating access requests [56].

Nevertheless, the separation between the abstract layer of policy specifications and their actual implementation seems to be a common problem in every kind of access control, and in every domain [110, 22]. Very often the two tasks are performed by different groups of people, having different background and using different languages and technology. It is also possible that some specifications come from legal requirements to which the policy must comply, and thus have been defined outside the organization itself. Communication among those groups is difficult and may cause misunderstandings. Usually this leads to poor security and efficiency [22]. Sometimes, the same group carries both the tasks of designing and implementing the security policy. But also in this case, these tasks are performed in two separate phases, and using different languages and level of abstractions. Indeed, using the same languages and models for both specification and implementation is not optimal, even when a single group takes care of both. As in any activity, security engineers should use the right level of abstraction for the task they are performing, otherwise there is the sensible risk to miss important interactions or details that lead to violations. It is well known that "the devil is in the detail," especially in security. The productivity and reliance of systems increases if the engineers can actually select which details are to be taken into account for each sub-task, at various steps of their work. But the division has to be done properly. In particular there must be no discrepancies between the different abstraction layers. Policy designer and configuration implementer operate on the same system using different representations that must be consistent. Passing from a layer to another is always error prone, hence reliable tools are needed.

Assume for example to have a network of terminals and servers connected to the Internet. When specifying who can access the services, the policy designers will use a representation that considers users, services, and classes of users and services. A possible requirement could be that "*users from the Internet cannot access critical services.*" The implementation may be a firewall configuration that drops every packets having an external IP address and a server address in the source and destination fields, respectively. Even in this very simple example it is not trivial to understand if the implementation is correct, because it heavily depends on low-level details. Indeed, consider an IP spoofing attack, a malicious

user from the Internet sends a packet to the server but using a forged source IP address. If the configuration is implemented without any countermeasure and mitigation for this kind of attacks, the firewall will let these packets pass even if it should not. However, it is reasonable that the high-level specification describes the policy without bothering about these low levels details. Ensuring that all the possible cases are considered when implementing the specification is tricky, especially because different requirements may interact. Furthermore, other interactions between the two abstraction layers have to be considered. Implementing the given specification and verifying the implementation with respect to a high level requirement is not always the end of the story. Security policies may evolve over time, hence updates and maintenance are also relevant. The attack surface is indeed different at the two levels: for this reason it is not always reasonable to assume that everything is to be managed at the high level. Hence, the problem of bridging the gap between these two layers is relevant for the management of access control configurations.

Two types of problems may occur when interfacing the two levels, some misunderstanding about the low level may happen when working with the high level or vice-versa. One of the common problems of the first type is that the abstract model, used to design the security policies, hardly takes into account the limitations that real systems have. These limitations could arise both when accessing information about the access requester and when enforcing the decision. Because of this mismatch of expressivity, sometimes the implementer may have to work with policies that cannot be enforced, forcing him to simplify, and sometimes also to resolve conflicts among different requests, which should instead be performed at the specification level. In practice the implementer would have to make policy-design decisions at deployment time, possibly without notifying policy designers, and getting a feedback from them.

Moreover, deploying a security policy requires to translate the high level specifications into low level configurations, expressed in implementation-dependent languages, which are possibly obscure due to legacy technology. Manually performing this task is often expensive and error prone since it requires the implementer to understand the exact correlation between the high level formulation and the low level constructs. Misbehaviours may result from a misunderstanding of the original policy or from a mistake due to intricacies of the low level language.

We present now some possible scenarios in which it is hard guaranteeing that nothing important is lost in between of the two levels of policy specifications and implementation.

Example 1.1. Suppose that in an organization different groups share a Unix server, in which each file is associated with an owner and the set of permitted operations to the group of the owner. At this level each user is part of a unique group, and it is not possible to make distinctions among different members of the same group.

Instead, it is common for people to be part of different groups. Moreover, not every member of a certain group has the same access rights, rather per-

missions are determined by the specific roles inside the group. The high level specifications are therefore role-based, whereas the low level is based on a more primitive mechanism: access control list with groups.

To actually implement the policy in the server, each subject may be associated with different users, and users may be grouped depending on a combination of groups and roles. The task of manually configuring the system to meet the requirements by creating the user accounts and groups is intricate, and even a small mistake can be difficult to spot. Moreover, updating and checking the correctness of the server configuration would be very tricky. For this reason these operations should be carried out at the high level, and somehow propagated accordingly to the low level.

Example 1.2. Consider a network that is protected by a firewall. Only some specific devices inside the network are allowed to connect to all the internal servers, whereas other servers are accessible by everybody. Moreover, users from the internal network may freely access the Internet.

Requirements are usually collected in informal languages that speak of abstracted entities like users, services and communication. The firewall conversely runs a configuration speaking of packets fields and interfaces. When implementing a specification it is not trivial to consider every possible case, with the risk of obtaining an over-permissive or over-restrictive policy.

The standard mechanism is based on by hand implementation plus testing for correcting misbehaviour. This is not the best approach since some corner cases can evade the testing, and more in general, manually coding the policy is not an efficient choice. Furthermore, the specification may also be impossible to implement, for example because it requires the firewall to realize a given policy of load balancing that is beyond the capabilities of the actual system. It is not a valid solution to ask the implementer to patch it, deciding how to downgrade the requirements without consulting the policy designers.

1.1.2 Our Proposal

We first give the desired properties that an approach must satisfy to correctly bridge the gap between specifications and implementation. Then we present three possible solutions for instantiating our two-layer approach in different real-world contexts. Finally, we discuss the challenges we face in order to succeed in our goal.

Desired Properties of the Two Layer Approach

We propose a methodology that includes fully-automated, formally-verified approaches for bridging the two abstraction layers, avoiding the waste of time and decrease of security when performing it by hand. At high level, policy designers know the limitations of the underlying system and their models should represent them correctly. Furthermore, the administrators in charge of deploying the policy take care of low level details only, possibly delegating an automated tool for translating the high level language into the one used by the actual system.

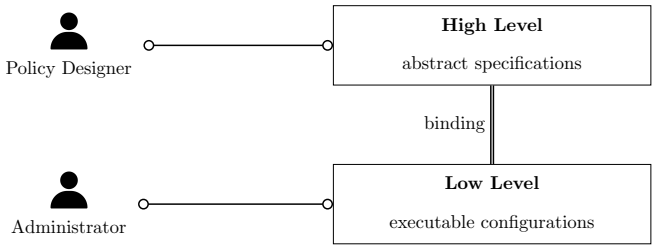


Figure 1.1: Our two-layers approach in a picture.

The types of languages used in the two abstraction layers usually follow completely different philosophies and styles. Moreover, they vary a lot, making it difficult to bridge among them. Clearly, high level languages for access control have the goal to ease the usage for general users. Some of them mimic natural language [95], others are based on graphical representations [97, 71]. Other proposals are based on formal logics [122], achieving a more precise semantics but also requiring the user to be confident with such formalism. Both ease of use and precision are indeed strong requirements.

Concrete languages for implementing access controls are usually domain specific, and largely depend on the underlying technology. Often they provide a limited capability of verifying conditions on the requester (for example, it may not be possible to make history dependent decisions, or to evaluate some class of attributes), as well as of performing actions on the requester and on the environment. In lots of cases, efficiency is a key element for a good configuration, because of the large amount of requests or low performance of the target device. Because of backward compatibility with legacy code, it is not unusual for state-of-the-art technologies to rely on very old and obscure languages that cannot be substituted with more adequate alternatives. A shallow example follows.

Example 1.3. Consider a corporate building, and assume that doors are controlled by smart readers (e.g., reading cards, PIN and fingerprints). The language used by the designers of security policy likely speaks of agents with certain attributes to be allowed or forbidden to access a room. Administrators work on one single lock at the time, with specific languages that predicate on attributes that must be satisfied or on raw access lists, depending on the available technology. Moreover the low level configuration may specify also how the doors interact with the requester, specifying the capabilities of each reader. In such a context, it may be the case that a policy could not be implemented as stated by the designers, and must undergo a revision phase by the administrators. Possibly, the low level layer also represents communications between the doors and the control panel that decides if the access has to be granted or denied.

Summarizing, there are (at least) two separate layers concerning access control (see Figure 1.1):

- the high level of conceptual policy design that abstracts from low level

implementation details and it is more suited for planning and specifying which behaviours to allow and which to prohibit;

- the low level in which configurations are implemented for the various entities that cooperate to realize the specification, each one with its own legacy language. Here the tuning of low level configurations must be taken into account, and practical limitations may occur.

The objective is to bridge between these separate layers. In practice, this serves to guarantee that the two different groups, namely, the policy designers and the system administrators, can communicate each other without errors.

Three Different Solutions

Different solutions may be applied to obtain this result. We group them into one-way translation, two-way translation and verification based.

One-way translation based solutions resort to compilation for automatically implementing the given specification that can be executed on the system. In this way, the administrator interacts with the system at the high level, delegating all the low level details to the compilation procedure. This is possible when the high level specification deals with all the details that cannot be automatically managed by the compilation.

Unfortunately, in some contexts, dealing with implementation details is a task that cannot be fully automatized (yet). Usually, this is because the high level abstracts from the low level details that need to be considered by human experts. Thus the administrator has to interact with the system at the low level, tuning or optimizing the executable configuration. In this case, both the low and high level representations of the system behaviour change, and if not managed correctly they can loose coherence.

A two-way translation solution works in this case, i.e., having both a compilation and a decompilation procedures to be applied when the specifications or the configuration change respectively. The administrator chooses the adequate level to interact with depending on the specific task, and the changes are propagated to the other level.

Verification based solutions do not rely on automatic updates and delegates to the user the task of updating one level when the other changes. If the requirements change the user has to manually update the low level configuration in order to meet the requirements; if the configurations change, the user has to manually update the high level specifications. To grant coherence, a procedure is given for checking if the two representations of the systems coincide. These solutions are adequate when the high level model abstracts lots of critical low level details that we still do not know how to manage automatically. In this case, using a compiled policy would be risky.

The choice between these three approaches is guided both by the complexity of the context where the access control operates and by the distance between the chosen high level and the actual system. Abstracting lots of details will resort in a more handy high level, but it increases the amount of work that

is delegated to automatic tools. Low level details are an important part of system administration, both for mitigating specific low-level attacks and for efficiency. At the current state of the art, sometimes automatic tools may be inadequate for this task. Because of this, low level details are taken into account in different ways by the three solutions: they are managed automatically in one-way translation, configured automatically and possibly modified by hand in two-way translation, and entirely in charge of humans in verification based solutions.

All three solutions guarantee coherence between the two representations and allow the policy to evolve over time to match the new requirements of the system.

Also expressivity limitations are considered. As previously stated, depending on the capability of the underlying system, some access control policies may be impossible to realize in practice. Expressing specifications that cannot be satisfied by the underlying system would be poorly useful, and even dangerous if administrator are unaware of the problem. Thus, we characterize the expressivity of the low level configuration language, and check if it matches the one of the high level language. When this is not the case, the compilation or verification procedure informs the policy designers, who can work out a solution when their requirements are beyond the capabilities of the underlying system.

The Challenges to Face

To apply the two-layer approach in the contexts we target, we have addressed different technical challenges.

First of all, we have to formally define the coherence between specifications and implementation. This is needed for proving both the correctness in verification based solutions and the semantic preservation of compilation and decompilation in translation based solutions. While the semantics of high level languages is usually well defined, for low level configuration languages an official formal specification is usually missing. Generally speaking, the meaning of configuration languages is given using examples and intuition, and some particular cases are not addressed at all, let alone the code itself, when available. Hence the definition of a formal semantics for the low level languages has been the first challenge in each domain that we target. We propose formal models of `iptables`, `pf`, `ipfw`, and SELinux CIL, and validate them through intensive testing, especially focusing on corner cases. We found several examples where the behaviour of the low level configuration languages is counterintuitive. Moreover, while formalizing SELinux CIL, we also discovered discrepancies between the observed behaviour and the official documentation, which has been signaled and verified by the developers (see Section 3.2).

A second concern is that it is usually algorithmically challenging to bind the two layers, especially when decompiling or verifying the specifications. When dealing with access control configurations, one can usually reduce to problems over finite sets only. However, explicitly enumerating all the possible access requests is not feasible in practice, because the number is huge. We resort to standard programming languages techniques, like model checking and SAT

solver algorithms. Indeed, these kinds of problems are specific instances of more general ones from various fields of computer science, but solutions are adjusted for the peculiar context of access control.

Finally, defining high level languages for specifying relevant policies is an interesting challenge per sé. We base our high level languages both on well known concepts, e.g., information flows in SELinux, and new paradigms, like contractual exchange policies in collaborative environments. In both cases, we design Domain Specific Languages aiming at ease of use. When formalizing new paradigms, we also develop non trivial semantics based on non-standard logic.

1.2 Three Solutions for Three Contexts

Now we briefly introduce the three different contexts that we target and discuss how we instantiate the two-layers approach to solve specific problems.

1.2.1 Networks Firewalls as Functions

In network contexts, we target firewalls. Firewalls are one of the most used tools for protecting computer networks and enforcing access control policies. They grant control on which packets can enter a network and how they are transformed by the so-called network address translation (NAT).

Roughly, low level executable configurations are lists of rules that transform and filter the incoming packets. Each firewall system comes with its own configuration language, with a specific syntax and a different way of evaluating and applying the rules. In addition, features like control-flow instructions and packet tagging mechanisms are used to better organize the configurations, but may also complicate understanding and management. Moreover, some rules may shadow others or prevent them from being triggered, depending on the order in which they appear in the configuration. This makes it hard to read and modify firewall configurations.

We formally model the most common firewall systems for Unix, i.e., `iptables`, `pf` and `ipfw`. We also study their expressive power and show that they are not equally expressive [10, 11, 9].

At the high level, we abstractly represent firewall policies as functions over IP packets, and propose a SQL-like query language for interacting with them. The policy designer can directly specify what to do with connection requests in a declarative way.

A two-way translation is obtained by proposing a compilation from functions to executable configurations and vice-versa. The administrators can interact with the system at the level they need for their specific task. In addition, the expressivity of configurations language is applied for checking if the high level specification can be implemented in the target system. A couple of tools are given and experimentally evaluated: FWS for compiling and decompiling that also supports the administrators with additional features, and F2F for comparing a policy with the expressivity of the low level configuration language.

In this particular instance of the two-layer approach, the low level language is completely transparent to the high level user. Indeed, the high level language is not based on the syntax of the configuration languages, but on their semantics only. When interacting with the abstract specification, the administrator is not required to know the underlying system. Only the semantics of configurations has to be considered, expressed as a function over packets. This work is discussed in Chapter 2.

1.2.2 Specifying Information Flow in Operating Systems

In the context of Operating Systems, we target Security Enhanced Linux (SELinux), a set of extensions of the Linux kernel that implements a Mandatory Access Control mechanism in Linux servers and in mobile devices [114]. SELinux configurations are conceptually simple: the system administrator defines a set of *types*, uses them to label all system resources and processes, and then defines the permitted operations of processes of each type. However, its use is far from being simple. Writing, understanding, and maintaining SELinux security policies is notoriously difficult as evidenced by numerous examples of misconfigurations in widely used policies [76]. SELinux developers recently proposed CIL (Common Intermediate Language), a promising declarative language that supports the definition of structured configurations, using, e.g., namespaces and macros.

Abstract policies for a system access control commonly predicate on permitted and prohibited information flows between OS entities [65, 48]. We propose *IFL* (Information Flow Language) a domain specific language (DSL) for expressing information flows requirements, including confidentiality, integrity, and non-transitive properties. The administrator express both the permissions that must be granted to users to perform their authorized tasks, as well as security requirements preventing dangerous operations on critical entities.

Summarizing, we identify the high level of the system with information flow specification, abstractly representing the expected behaviour of the system with functional and security requirements. The low level is instead the one of CIL configurations, where every single permission is listed for each entity.

The binding between the two layers is obtained by IFCIL, an extension of CIL supporting information flow requirements, and a verification procedure for statically checking that a configuration satisfies its requirements. Intuitively, the languages of the two layers are merged, and the operators of CIL acts on IFL requirements as with native CIL instructions. The different layers are adequate at performing the different tasks: the administrator interact either with the system specification by operating on the IFL requirement, either with the executable CIL configuration, and the verification procedure ensures that the two layers are coherent. The verification procedure is implemented in the tool IFCILverif that we applied on real-world configurations available online. This work is discussed in Chapter 3.

1.2.3 Permission Exchanges in Collaborative Environments

In the context of collaborative environments we target the problem of granting a mutual benefit when exchanging access rights. In a distributed setting, each user has a set of his own resources that are possibly shared with others. Resources may be either infinite (or reusable), where the asset is still available to the owner after he allows access to it, or finite, i.e. that are consumed when exchanged. The access policy protecting these resources should aim at mutual advantages. For example, different hospitals may share anonymized medical data to improve the quality of statistics.

Traditional access control languages do not express conditions that foster mutual benefits, but only check the roles or the attributes. Typically, the exchange of access rights and resources is negotiated by humans and implemented by hand. We propose *MuAC*, a new high level policy language where access control decisions are based on permission exchanges between the requester and the owner of the resource. Polices of different users impact each other, even though they are defined in isolation and only control the access to the resources of a single user.

For the low level, we do not work on existing access control systems. Instead, we propose a logical low level. MuAC relies on the contractual reasoning typical of human agreements. Unfortunately, this kind of reasoning may induce circularity while evaluating access requests, that are not correctly managed by classical logic. To overcome this limitation, we introduce an adequate non-standard logic for contracts.

For binding the abstraction layers, we give a compilation from MuAC policies to logical theories, that grants the coherence between the high level and low level representations of the system.

Notably, since this time the low level is not a preexisting access control technology, we propose an implementation based on blockchain smart contracts, and an off-chain client. Those allow the user to interact with the system at the high level abstracting away low level details that are entirely in charge of the proposed compilation and implementation. This work is discussed in Chapter 4.

1.3 State of the Art

Access control is a widely studied field, surveyed by many papers, e.g. [100, 112, 102]. Over the years, different models for defining access controls policies have been proposed. Traditional models specifically identify subjects and resources. They assign access rights directly enumerating the allowed operations for each pair of subjects and resources. A general approach for describing this assignment is an access matrix [83]. Rows of the matrix consist of subjects, columns of resources that may be accessed. Each entry in the matrix indicates the access rights of a particular subject for a particular resource. In practice, an access matrix is usually sparse and is represented as a set of columns, yielding access control lists (ACLs), or rows, yielding capability tickets. For each object, an

ACL lists users and their access rights. Conversely, for each user, a capability ticket specifies authorized objects and operations. A common extension of these models consider groups. Each user is thus inserted in a specific group and acquires access rights, accordingly.

Traditional systems define the access rights of individual users and groups of users. In contrast, Role Based Access Control (RBAC) is based on the roles that users assume [53]. Roles typically represent job functions within an organization. This model is the standard from medium to large organizations. Rights are assigned to roles instead of subject, allowing more flexibility and ease of configuration with respect to the previous model. Users are assigned roles, either statically or dynamically. When the assignment is dynamic, the mapping between subjects and roles is temporary, i.e., last one session. However, despite its multiple variants RBAC hardly permits the specification of fine-grained controls. In fact, introducing fine-grained authorizations leads to a proliferation of roles, and hence to serious problem when maintaining the system over time.

A relatively recent development in access control technology is the Attribute-Based Access Control (ABAC) model [73]. In the ABAC model authorizations are based on properties of both the resource and the subject. Rules are based on attributes, i.e. arbitrary security-relevant information exposed by the system, the involved subjects, the action to be performed, or by any other entity of the evaluation context that are relevant to the rules in hand. The strength of the ABAC approach is its flexibility and expressive power. Indeed it allows the administrator to choose the level of detail he wants when writing policies: he may use very specific attributes like username, or very generic ones like role or age. Concerns about its performance have been the main obstacle against its adoption [21]. However, in contexts like Web services and cloud computing, this increased performance cost is less noticeable because each access request has often a high cost. An important step for the spread of ABAC has been the introduction of the eXtensible Access Control Markup Language (XAMCL)[2]. Here, the rules are based on arbitrary information from the system and organised in structured collections called *policies*.

There is an additional orthogonal classification of access control models that is based on the ownership of the controls: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). In the former, the controls belong to the owner of the resource, while in the latter the system controls the access and the resource owner cannot circumvent them.

Access control is a very broad research area that spans from physical security to network security. Some main aspects are common to every domain, whereas there are a lot of details that depend on the specific context. Below, we will talk generally when possible, explicitly specifying when we do not.

Access control systems have attracted the attention of researchers in computer security since long time. Among the several proposals, we only focus on language-based approaches, in particular on works where the abstract policy of the system is expressed in a high level language and the actual behaviour in a low level configuration language. This idea of having two separate languages is actually quite widespread. Usually the contribution of the considered papers

can be described as the propagation of some information from one layer to the other, thus achieving something similar to what we advocate in Section 1.1. In some cases, an explicit translation among the two languages is given, other times more specific tasks are considered instead, like policy verification or visualization.

A rough classification groups the works similar to our in three families. In the first, the emphasis is on the high level specification. Some results are then reflected in the low level system. In this family, a typical goal is the synthesis of a low level policy, through some sort of compilation. In the second class, we put those proposals that start from the low level system. Their aim is to verify some high level properties, or to extract an abstract representation of the behaviour of the system. Finally, the third class has approaches that combine the two lines above.

We start by discussing papers in the first class. In general the intended objective is allowing the security engineer to manage the system at high level only [122, 14, 21].

Tsankov et al. [122] define global access policies for physical spaces, expressed using CTL formulae over paths inside the building. Then, the problem of synthesizing a local attribute-based policy for each door or turnstile is formally defined and proved NP-hard. An algorithm based on Satisfiability Modulo Theories (SMT) solvers is presented and evaluated against three real configurations and several artificially generated ones, showing good performances. Calo et al. [30], investigate the self-generation of access control policies as an alternative to manual configuration when the environment is very dynamic and the device is capable of assessing the variation of the environment. The general idea is that the administrator manually describes either the overall behaviour, or the dangerous states to prevent. Using this description and its own perception of the environment, the system then generates the access policy, e.g., using machine learning. Several dynamic access control approaches are presented, considering different degrees of autonomy.

ABAC can be considered a generalization of RBAC, since attributes can indeed model roles. Hence, ABAC languages allow more flexibility and may thus be preferable for the specification of security requirements. In contrast, RBAC systems are more efficient and are de facto standard in organizations. The translation proposed in [21] allows deploying ABAC policies in RBAC systems, actually taking the best from both: the flexibility of ABAC and the efficiency of RBAC.

In cloud computing platforms, one can distinguish between security mechanisms "of the cloud", i.e., implemented and managed by the cloud provider, and security mechanisms "in the cloud", i.e., offered by the cloud provider to the customer to be configured with the desired access control policy. Well-engineered security mechanisms are available in different cloud platforms, but they are not simple to configure and depend on the chosen cloud provider. Moreover, the security of cloud applications across different platforms is difficult to assess. Morelli et al. [92] propose a high level language that abstract away from specific enforcement mechanisms of the cloud platform and is compiled to

policies for two of the most widely used platforms: AWS [1] and Openstack [3]. The high level language allows the customers to express their policies using a simplified structured natural language based on the ABAC approach, that subsumes the mechanisms offered by the two providers. A formal semantics of the high level language is given, as well as an encoding in first order logic and a SMT-based tool for proving security properties before compiling. Expressivity limitations of the underlining access control systems are also considered: cloud platforms are not able to fully support the complexity and granularity of the proposed high level language. When the compilation is not possible, the tool reports the skipping of the components not supported/recognized within the specific platform.

In collaborative systems, several different entities prescribe access decisions for the assets. Their policies may be in contrast, e.g., when a request is to be served according to the policy of one entity and rejected according to another. To allow the system to make a decision, conflicts must be resolved. Resolution may be, for example, based on a hierarchy among the entities. The hierarchy is based on the importance of the relationship that the different entities have with the asset. In this case higher entities in the hierarchy are privileged, as in [46]. Den Hartog et al. [46] extend this approach, by keeping for each request the part of the hierarchy that determines the final decision, and by notifying the user when its policy is not enforced, carefully justifying the choice. Indeed, also in collaborative systems it is quite common to have two level of representation [46, 47, 107, 29]. In the high level, several policy exists, one for each different entity, whereas in the low level there is only the combined policy.

Also in the domain of firewalls, many papers take a top-down approach, proposing ways to specify abstract filtering policies that can be possibly compiled into the actual firewall systems, e.g., [14, 15, 45, 61, 18, 55, 19]. Some of these approaches only consider the simplest types of rules [14, 61, 55, 19]. This is also because of the enormous amount of extensions, modules and utilities of commercial products, which are rarely and poorly documented. Probabilistic rules, Network Address Translation (NAT), load balancing options and stateful policies are often totally or partially ignored by formal tools. Therefore, one cannot avoid considering low level languages at all. An interesting approach in [50], a formal framework for reasoning on `iptables`. In case some feature is encountered that is not considered in the formal model, the behaviour of the system is explicitly approximated in a controlled manner. In particular, the authors provide a "cleaned" ruleset that an automatic tool can easily analyze, using a formal semantics of `iptables` mechanized in Isabelle/HOL [98].

We now consider works in the second family. Rarely a full decompilation is even taken into account. As a matter of fact, usually, specific tools exist for each different task, and only the needed information is translated to the high level. Some works aim at verifying arbitrary properties expressed with quite general formalism, others just focus on checking for well-known errors or visualize the configuration semantics in human readable ways. In general, it is not always possible to apply those results to check if a modified configuration still satisfies the requirements.

We first consider the domain of physical access control. O'Sullivan [99] describes both the topology of a building and requirements about reachability of rooms, then he checks the presence of a number of predefined anomalies using a SAT solver. Access Nets are introduced in [57] for reasoning about properties of physical spaces with pointwise access control mechanisms. A SMT-based tool is presented to check reachability of rooms from subjects. Also temporal constraints are considered. Given a Building Information Model description of a physical space [84], the tool of [113], derives a 3D representation of the behaviour of an access control policy, thus helping the administrator to understand the actual impact of possible changes. Topology aware adaptive security is applied in [124] to prevent a system from entering dangerous states, identified by a high level description based on a CTL formula. The low level modeling is very detailed. Besides paths inside the building, agents position and motion are explicitly modeled using the Ambient Calculus [31]. The same kind of task is performed by [123] on systems characterized by an interplay between cyber and physical elements, using Bigraphical Reactive System [91]. Given an ABAC policy, Rao et al. [106] extract a high level representation based on multi-level grids. Several works on ABAC policy analysis target XACML [7], a de-facto standard. Among them, [125, 126] address the problem of analyzing policies with non-boolean variables and functions through a SMT solver. They propose a query language that allows the specification and analysis of a vast range of security properties that have been proposed in the literature (attribute hiding, separation of duty, policy refinement, policy subsumption, change impact, scenario finding). A proved correct translation is proposed from XACML policies into SMT formulas. Types and functions available in XACML are thus mapped in the theories offered by the SMT solver. Decidibility is informally discussed showing that quantifier-instantiation procedures are usually successful, because of the specific form of the formulas used for XACML policy analysis. Also the complexity is discussed by taking into account the individual complexity of the theories and how they compose, which often lead to NP-hard problems. Despite the high theoretical complexity, the SMT solver performs well in the experiments, and the authors argue that this depends on the kind of formulas derived from XACML policy analysis. The proposed approach is compared with similar ones that use different SAT solvers, showing a major improvement in terms of both memory usage and computational time.

As regards firewalls, Jayaraman et al. [78] propose an approach for validating network connectivity policies, implemented by the tool SecGuru. They extract logical specifications from real Cisco routers and solve them in Z3.

Few papers consider a bidirectional binding between the two layers, implementing both compilation and decompilation among the two languages. Margheri et al. propose FACPL [85], a formally-defined, fully-implemented ABAC language. It closely resembles XACML, actually, it is possible to compile FACPL into XACML and vice-versa. Moreover, the authors use a SAT-based tool for verifying properties of a single policy and of the relationships among multiple policies.

Some papers propose high level languages with new features for specific prob-

lems. They rarely deal with real implementations. Some focus on Relationship Based Access Control (ReBAC) [56], others address the problem of collaborative systems and conflict resolution [89], or delegation and revocation [42].

Several works consider the expressivity of different classes of access control languages [24, 16, 43, 107]. Some of these also consider fragments of real world languages like XACML [43, 107].

1.4 Structure of the Thesis

In Chapter 2 we present our work about networks, which allows to configure and update a firewall both at the high level of abstract policies and the low level of executable `iptables`, `pf`, and `ipfw` configurations. We implement both a compiler and decompiler, and study the expressivity of the considered languages. In Chapter 3 we target operating system security, extending the low level of SELinux configurations with high level features for specifying permitted and prohibited information flows. We also give a verification procedure for checking that requirements are satisfied by the low level. In Chapter 4 we consider collaborative environments, and propose a new access control language for expressing policies for exchanging access rights. We propose a logical low level and a compilation from high level policies to logical theories, as well as a procedure for deciding requests and a blockchain implementation. Finally, in Chapter 5 we conclude and in Appendix A, B, and C we give the formal details and the proofs of Chapter 2, 3 and 4 respectively.

1.5 Published Work

Hereafter we briefly present the candidate's publications containing parts of the thesis.

From Firewalls to Functions and Back by L. Ceragioli, L. Galletta, M. Tempesta. [39]

This work present a preliminary proposal for dealing with firewall policies represented as functions. It contains preliminary parts of Chapter 2.

Are All Firewall Systems Equally Powerful? by L. Ceragioli, P. Degano, L. Galletta. [35]

This paper analyzes the expressive power of different firewall languages for Unix-based systems. The two different kinds of expressivity proposed in Chapter 2 are defined.

Checking the Expressivity of Firewall Languages by L. Ceragioli, P. Degano, L. Galletta. [36]

We propose F2F, a tool that relies on the results of the previous work to check if a given firewall policy can be implemented in or migrated to a given firewall language. The tool is discussed in Chapter 2

FWS: Analyzing, maintaining and transcompiling firewalls by C. Bodei, L. Ceragioli, P. Degano, R. Focardi, L. Galletta, F. L. Luccio, M. Tempesta, L. Veronese. [26]

We propose FWS, a tool that allows to analyze, maintain and transcompile firewall configurations. It implements compilation and decompilation from executable configurations and functions over packets represented as tables. The modelization of Unix firewalls, and the technical details about compilation and decompilations of Chapter 2 are discussed.

Can my firewall system enforce this policy? by L. Ceragioli, P. Degano, L. Galletta. [38]

Several tools allow the user to specify firewall policies in various high level languages, and to compile them into different target configuration languages. We apply F2F to check when such a compilation is possible, giving detailed information about the unexpressible parts of a policy and providing administrators with hints for fixing the detected problems. It contains parts of Chapter 2.

MuAC: Access Control Language for Mutual Benefits by L. Ceragioli, P. Degano, L. Galletta. [37]

This work present a preliminary version of Chapter 4 that relies on a pre-existing logic and does not target finite resources, nor presents the blockchain implementation we propose here.

Chapter 2

Network

In this chapter we target network security and address firewalls. Firewalls are one of the most used tools for protecting computer networks and enforcing access control policies. They grant control on which packets can enter a network and how they are transformed by the so-called network address translation (NAT). The firewall behavior is specified by a configuration that implements the wanted policy. Roughly, a configuration is a list of rules that transforms and filters the incoming packets. Each firewall system comes with its own configuration language, with a specific syntax and a different way of evaluating and applying the rules. We consider the most common firewall systems for Unix, i.e., `iptables`, `pf` and `ipfw`. Different firewall languages can express different policies, and have indeed different sets of advanced features [10, 11, 9]. As a matter of fact, different systems have different expressive power, even only considering the very basic actions (i.e. accepting, dropping and performing NAT) [35].

Firewall languages usually provide control-flow instructions like `call` and `goto`, and a tag system that allows labeling the packets. These features may be used to better organize the configurations, but may also complicate their understanding and management. Moreover, some rules may shadow others or prevent them from being triggered depending on the order in which they appear in the configuration. This context-dependency makes it hard to read and modify firewall configurations.

Conversely, specifications usually put requirements on such abstract entities like users, services and connections. The specification language has to be easy to read, and should hide implementation details.

Summarizing, the two abstraction layers we work with are:

- the low level (LL), the one of firewall systems for Unix, i.e., `iptables`, `pf`, and `ipfw`, where the network and packets are represented in full detail, and in which firewall configurations are written in their specific language predicating over fields of packets and network interfaces;
- the high level (HL), the one of policy specifications, that allows the administrator to interact with the semantics of the firewall in an abstract

way, directly specifying what to do when users request connections to services, the options are either to discard or to allow them, and possibly masquerading the real requester or redirecting the request.

Note that both layers are useful for the specific task they serve. On the one hand, the low level configuration language is not suited for defining the specifications of the system, thus making it difficult to understand if the configuration is adequate to the requirements of the network. Actually, this language enable the administrator to explicitly choose how the firewall should manage the packets, hence increasing the efficiency.

On the other hand, high level languages are not usually designed for managing details at the level of packets, and can hardly support every single feature of the real configuration languages. Instead they allow the designer to grasp the emergent behaviour, and to specify and update the requirements in a readable but precise way.

It is thus desirable to allow the administrators to interact with the system at the level they need for their specific task, granting coherence between the different representations, as depicted in Figure 2.1. In particular, this is obtained with a bidirectional compilation-decompilation relation that given a low level configuration returns an abstract representation for the high level and vice-versa. We have already stated that different firewall systems have different expressive power. Hence the specifications may be impossible to implement in the chosen configuration languages. At the high level, the user must be informed of the limitation of the underling system. In practice, the compiler should inform the policy designer when the specification is not expressible, who can explicitly decide either to change the target system, or to rewrite the specification.

To prove coherence between the two representation, i.e. correctness of the bidirectional translation, it is important that both low and high level language are enriched with a formal semantics. Low level languages are thus modeled through a common modeling language, called IFCL, which has a formal semantics that is thus inherited by the Unix firewall languages. For the high level, we propose *FWQL* a SQL-like query language that operates on functions from packets to transformations, abstractly representing firewall configurations. The high level query language allows to visualize, update, and check the semantics of the firewall. Updating the specification by removing, adding or modifying the requirements is critical, because the use of the network may evolve over time.

Once the specification is compiled to an executable configuration, this can be reworked by system administrators to improve efficiency and readability. Refactoring may consist in the creation of specific rulesets for specific tasks, or in reordering the rules in a meaningful way. Moreover, the administrators may adjust low level details, like the fields that are used for assessing the legitimacy of a request. Our proposal is compatible with tools that help the administrator in these tasks.

Note that, in this particular instance of the two-layer approach, the low level language is completely transparent to the high level user. When interacting with the abstract specification, the administrator is not required to know the

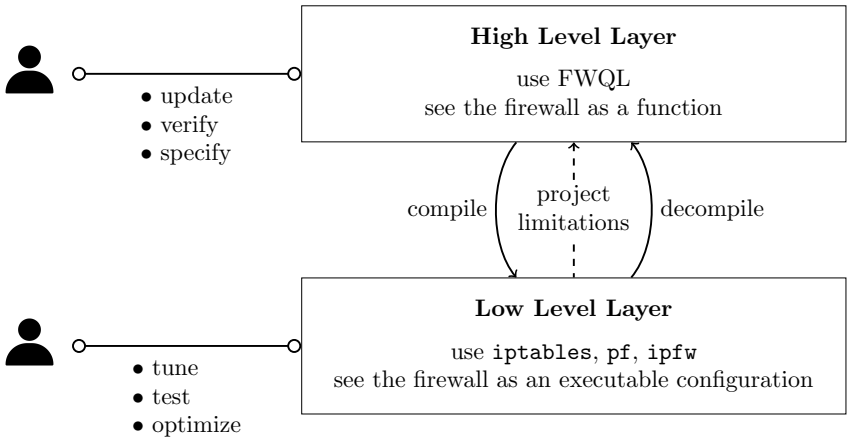


Figure 2.1: Schema of the two-layer approach for firewalls.

underlying system. Only the semantics of configurations has to be considered, expressed as a function over IP packets.

We implement our proposal and we propose two tools: FWS, that realizes the compilation and decompilation functions, and *F2F*, that checks of a given policy can be expressed in the chosen low level language. We validate both the tools on real-world configurations, showing that they operates with reasonable time cost.

Structure of the chapter

In Section 2.1 we introduce the firewall configuration languages we consider. In Section 2.2 we present IFCL and show how it encode them. In Section 2.3 we present FWQL, and show how it works on functions representing the firewall semantics. In Section 2.4 we show the decompilation of a firewall configuration to a functional representation. In Section 2.5 we present the compilation from functions over packets to executable firewall configurations. In Section 2.6 we characterize the expressivity of different firewall languages, and show how to check if the desired policy can be implemented in the target system. In Section 2.7 we present our tools, the first one for compiling and decompiling, the second one for checking the expressivity of firewall languages. In Section 2.8 we compare our proposal with similar works. Finally, we conclude in Section 2.9.

2.1 Background

Usually, system administrators classify networks into security domains. Through firewalls they monitor the traffic and enforce a predetermined set of access control policies among the various hosts and subnetworks (*packet filtering*). System administrators can also use a firewall to connect a network with private IPs to

other (public IP) networks or to the Internet and to perform connection redirections through Network Address Translation (NAT).

Firewalls are implemented either as proprietary, special devices, or as software tools running on general purpose operating systems. Independently of their actual implementations, they are usually characterized by a set of rules that determine which packets reach the different subnetworks and hosts, and how they are modified or translated.

Below, we briefly review three different firewall systems: `iptables` [10], `ipfw` [9] and `pf` [11] that are the most used in Linux and Unix. We refer the reader to the manuals of these systems for additional details. In the following, we abstract away from the specific syntax of these languages. Nevertheless, the reader can find examples of simple configurations in subsection 2.7.1.

`iptables`

It is one of the most used tools for packet filtering being the default in Linux. It operates on top of Netfilter, the packets processing framework implemented in the Linux kernel [109].

The basic notions of `iptables` are *tables* and *chains*. Intuitively, a table is a collection of ordered lists of policy rules called chains. The most commonly used tables are: `filter` for packet filtering; `nat` for network address translation; `mangle` for packet alteration. There are five built-in chains that are inspected at specific moments of the packet life cycle [119]: `PreRouting`, when the packet reaches the host; `Forward`, when the packet is routed through the host; `PostRouting`, when the packet is about to leave the host; `Input`, when the packet is routed to the host; `Output`, when the packet is generated by the host. Tables do not necessarily contain all the predefined chains and further user-defined chains can be present.

Chains are inspected top-down to find a rule applicable to the packet under elaboration. Each rule specifies a condition and a target: if the packet matches the condition then it is processed according to the specified target, which can be either a built-in target or a user-defined chain. The most commonly used targets are: `ACCEPT`, to accept the packet; `DROP`, to discard the packet; `RETURN`, to stop examining the current chain and resume the processing of a previous chain; `DNAT`, to perform destination NAT, i.e., a translation of the destination address; `SNAT`, to perform source NAT, i.e., a translation of the source address. When the target is a user-defined chain, two "jumping" modes are available: *call* and *goto*. The difference between the two arises when a `RETURN` is executed or the end of the chain is reached: the evaluation resumes from the rule following the last matched call (just as for standard call-return of procedures). Some extensions also allow marking packets with a tag and to verify the tag value. Built-in chains have a user-configurable default policy (`ACCEPT` or `DROP`) which determines the future of a packet when no rule applies.

`ipfw`

It is the standard firewall for FreeBSD [9]. A configuration consists of a single ruleset that is inspected twice, when the packet enters the firewall and before it exits. It is possible to specify whether a rule should be applied only in one of the two directions using the keywords `in` and `out`.

Similarly to `iptables`, rules are inspected sequentially until the first match occurs and the corresponding action is taken. The packet is dropped if there is no matching rule. The most common actions in `ipfw` are the following: `allow` and `deny` are used to accept and reject packets; `nat` applies destination NAT to incoming packets and source NAT to outgoing packets; `check-state` accepts packets that belong to established connections; `skipto`, `call` and `return` allow to alter the sequential order of inspection of the rules in the ruleset. Differently from `iptables`, the targets of `skipto` and `call` are rules instead of rulesets. A packet is dropped if there is no matching rule.

Packet marking is supported also by `ipfw`: if a rule containing the `tag` keyword is applied, the packet is marked with the specified identifier and then processed according to the rule action.

`pf`

This is the standard firewall of OpenBSD [11] and is included in macOS since version 10.7. Each rule consists of a predicate which is used to select packets and an action that specifies how to process the packets satisfying the predicate. The most frequently used actions are `pass` and `block` to accept and reject packets, `rdr` and `nat` to perform destination and source NAT. Packet marking works as in `ipfw`. Differently from the other firewalls, the action taken on a packet is determined by the *last matched rule*, unless otherwise specified by using the `quick` keyword. There is a single ruleset in `pf` that is inspected both when the packet enters and exits the firewall. As in `ipfw`, keywords `in` and `out` can be used to specify whether a rule should be applied only in one direction. When a packet enters the host, DNAT rules are examined first and filtering is performed after the address translation. Similarly when a packet leaves the host: first its source address is translated by the relevant SNAT rules, and then the resulting packet is possibly filtered. Packets belonging to established connections are accepted by default, thus bypassing the filters.

2.2 Formalizing the Low Level

We define a common intermediate language for modeling firewall systems called IFCL. IFCL is enriched with a formal semantics and subsumes the languages that we target. First, we present IFCL and show how it encode a specif firewall configuration in `iptables`, `pf`, or `ipfw`; then we give its semantics and a normalization procedure for removing control-flow instructions.

2.2.1 Intermediate Firewall Configuration Language: IFCL

Our intermediate firewall configuration language IFCL is parametric with respect to the steps performed by the system to elaborate packets. Let $\mathbb{P}$ be the set of IP packets, formally the Cartesian product of domains D_w, one for each possible field of a packet. For generality, we do not detail the format of network packets, we only assume W to be finite and to contain the fields $dIP, dPort, sIP$ and $sPort$ that are the destination (source) IP and port, respectively. In the following we use $sa(p)$ and $da(p)$ to denote the source and destination addresses of a given packet p. An address a consists of an IP address $IP(a)$ and possibly a port address $port(a)$. An *address range* n is a pair consisting of a set of IP addresses and a set of ports. Address a (and address ranges n) are sometimes written explicitly as $IP(a)$:$port(a)$ (and $IP(n)$:$port(n)$). An address $a = IP(a)$:$port(a)$ is in the range $n = IP(n)$:$port(n)$, written $a \in n$, if $ip(a) \in ip(n)$ and $port(a) \in port(n)$ when $port(a)$ is defined, e.g., for TCP and UDP packets. Usually, firewall languages allow to mark packets with tags. Such tags are not propagated outside the firewall itself, and are only used to match the packets inside the rulesets. We write $tag(p)$ for the tag m associated with p, assuming that the empty tag $\bullet$ identifies untagged packets.

To record the modifications on packets, we write $p[da \mapsto a]$ to denote a packet identical to p except for the destination address da which becomes equal to a. Similarly, $p[sa \mapsto a]$ denotes a packet p where the source address is a and $p[tag \mapsto m]$ denotes p with a modified tag m.

Here, we consider stateful firewalls that keep track of the state s of network connections and use this information to process a packet. An existing network connection is described by several protocol-specific properties, e.g., source and destination addresses or ports, and by the translations to apply. In this way, filtering and translation decisions are not only based on administrator-defined rules, but also on the information built by previous packets belonging to the same connection. We omit a precise definition of a state, but we assume that it tracks at least the source and destination ranges, NAT operations and whether a connection is established or not. When receiving a packet p one checks whether it matches the state s or not. We left unspecified the match between a packet and the state because it depends on the actual shape of the state. When the match succeeds, we write $p \vdash_s \alpha$, where α describes the actions to be carried out on p; otherwise we write $p \nvdash_s$.

A firewall rule is made of two parts: a predicate ϕ expressing criteria over packets, and an action t, called *target*, defining the destiny of matching packets. Just as in real languages, predicates are formulas in a propositional logic that mainly express identity or inclusion between ranges of IP addresses or ports, as well as the protocol in usage and the current state of the firewall. Hereafter, we shall write $\phi(p)$ whenever the fields of the packet p satisfy the constraints expressed by the proposition ϕ. Here we only consider a core set of actions included in most of the real firewalls. These actions not only determine whether or not a packet passes across the firewall, but they also control the flow in which

the rules are applied. They are:

ACCEPT	let a packet pass
DROP	discard a packet
CALL(R)	invoke the ruleset R (see below)
GOTO(R)	jump to the ruleset R
RETURN	exit from the current ruleset
NAT(n_d, n_s)	apply address translation
MARK(m)	mark with tag m
CHECK-STATE(X)	examine the state

The targets CALL(_) and RETURN implement a procedure-like behavior; GOTO(_) is similar to unconditional jumps. We only have the action DROP and not a REJECT, because for our analysis there is no need of keeping rejected and dropped packets apart. In the NAT action n_d and n_s are address ranges used to translate the destination and source address of a packet, respectively; in the following we use the symbol $\star$ to denote an identity translation, e.g., $n : \star$ means that the address is translated according to n, whereas the port is kept unchanged. The MARK action marks a packet with a tag m. The argument $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ of the CHECK-STATE action denotes the fields of the packets that are rewritten according to the information from the state. More precisely, $\rightarrow$ rewrites the destination address, $\leftarrow$ the source one and $\leftrightarrow$ both.

Definition 2.1 (Firewall rule)**.** A firewall rule r is a pair (ϕ, t) where ϕ is a logical formula over a packet, and t is the target action of the rule.

A packet p matches a rule r with target t whenever ϕ holds.

Definition 2.2 (Rule match)**.** Given a rule $r = (\phi, t)$ we say that p matches r with target t, denoted $p \Vdash_r t$, if and only if $\phi(p)$. We write $p \nVdash_r$ when p does not match r.

We now define how a packet is processed given a possibly empty list of rules (denoted with ϵ), called *ruleset*. Similarly to real implementations of firewalls, we inspect the rules in the list, one after the other, until we find a matching one, which establishes the target of the packet. For sanity, we assume that no GOTO(R) and CALL(R) occur in the ruleset R, so avoiding self-loops. We also assume that rulesets have a default target denoted by $t_d \in \{\text{ACCEPT}, \text{DROP}\}$, which accepts or drops according to the system administrator's decision.

Definition 2.3 (Ruleset match)**.** Given a ruleset $R = [r_1, \ldots, r_n]$, we say that p matches the i-th rule with target t, denoted $p \Vdash_R (t, i)$, if and only if

$$r_i = (\phi, t) \wedge p \models_{r_i} t \wedge \forall j < i \,.\, p \nVdash_{r_j}.$$

We also write $p \nVdash_R$ if p matches no rules in R, formally if $\forall r \in R \,.\, p \nVdash_r$.

In our model we do not explicitly specify the steps performed by the kernel of the operating system to process a single packet passing through the host. We

represent this algorithm through a *control diagram*, i.e., a graph where nodes represent different processing steps and the arcs determine the sequence of steps. The arcs are labeled with a predicate describing the requirements a packet has to meet in order to pass to the next processing phase. Therefore, they are not finite state automata. Our control diagrams are deterministic, i.e., every pair of arcs leaving the same node has mutually exclusive predicates. For generality, we let these predicates abstract, since they depend on the specific firewall.

Let Ψ be a set of predicates over packets, assuming that for all $\psi \in \Psi$ and for all $p \in \mathbb{P}$, $\psi(p) = \bigwedge_{w \in W} \psi_w(p_w)$.

Definition 2.4 (Control diagram)**.** A *control diagram* $\mathcal{C}$ is a tuple (Q, A, q_i, q_f), where

- Q is the set of nodes;
- $A \subseteq Q \times \Psi \times Q$ is the set of arcs with no self-loops and such that whenever (q, ψ, q'), $(q, \psi', q'') \in A$ and $q' \neq q''$ then $\forall p.\, \neg(\psi(p) \wedge \psi'(p))$;
- q_i,$q_f \in Q$ are special nodes denoting the start and the end of elaboration.

The firewall filters and possibly translates a given packet by traversing a control diagram accordingly to the following transition function.

Definition 2.5 (Transition function)**.** Let (Q, A, q_i, q_f) be a control diagram and let p be a packet. The transition function $\delta \colon Q \times Packet \mapsto Q$ is defined as

$$\delta(q, p) = q' \quad \text{if and ony if} \quad \exists (q, \psi, q') \in A.\ \psi(p) \text{ holds.}$$

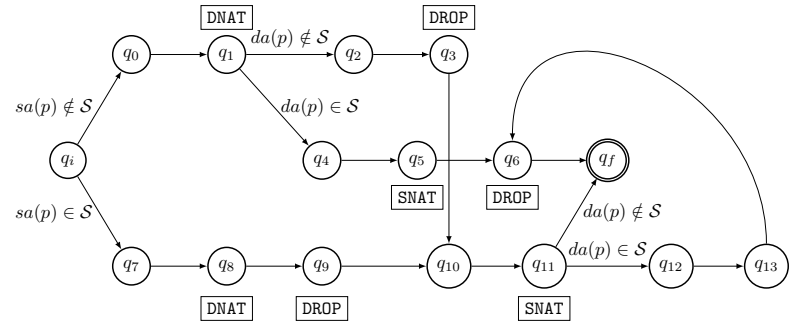
A firewall in IFCL is defined as follows.

Definition 2.6 (Firewall)**.** A firewall $\mathcal{F}$ is a triple $(\mathcal{C}, \rho, c)$, where $\mathcal{C}$ is a control diagram; ρ is a set of rulesets; and $c \colon Q \mapsto \rho$ is the *correspondence* mapping from the nodes of $\mathcal{C}$ to the actual rulesets.

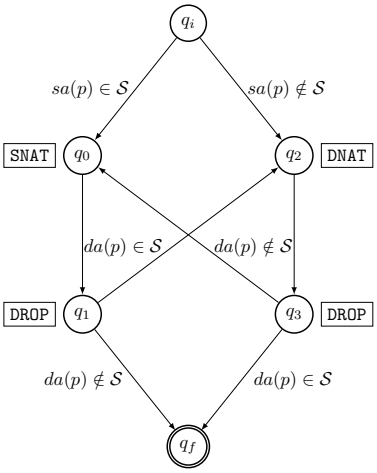
Note that the set of rulesets ρ, and the correspondence mapping c depend on the specific firewall configuration, whereas the control diagram of a firewall $\mathcal{C}$ only depends on the firewall system. In the following, we will write $\mathcal{C}_{\mathcal{L}}$ for the control diagram of the firewall language $\mathcal{L} \in \{\texttt{iptables}, \texttt{pf}, \texttt{ipfw}\}$.

2.2.2 Encoding Unix Firewalls

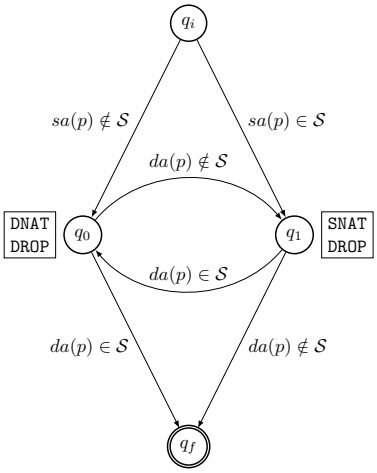
Here we encode the firewalls system considered so far as triples $(\mathcal{C}, \rho, c)$ of our framework. Notably, the encoding provides a formal semantics for those systems defined in terms of IFCL. In the following, let $\mathcal{S}$ be the set of local addresses of a host, i.e., those associated with its network interfaces. The control diagrams of `iptables`, `pf` and `ipfw` are in Figure 2.2, where unlabeled arcs carry the label "*true*," and labels `DNAT`, `SNAT` and `DROP` are associated with nodes capable of modifying destination or source addresses and discarding packets respectively (they will be used in Section 2.5 and Section 2.6).



(a) Control diagram of `iptables`



(b) Control diagram of `pf`



(c) Control diagram of `ipfw`

Figure 2.2: Control diagrams of `iptables` and `ipfw`.

iptables Figure 2.2a shows the control diagram $\mathcal{C}_{\texttt{iptables}}$ of `iptables`. It also implicitly defines the transition function according to Definition 2.5. In `iptables` there are twelve built-in chains, each of which correspond to a single ruleset. So we define the set $\rho_p \subseteq \rho_{\texttt{iptables}}$ of primitive rulesets as the one made of R^{man}_{INP}, R^{nat}_{INP}, R^{fil}_{INP}, R^{man}_{OUT}, R^{nat}_{OUT}, R^{fil}_{OUT}, R^{man}_{PRE}, R^{nat}_{PRE}, R^{man}_{FOR}, R^{fil}_{FOR}, R^{man}_{POST} and R^{nat}_{POST}, where the superscript represents the chain name and the subscript the table name. Needless to say, the set $\rho_{\texttt{iptables}} \backslash \rho_p$ contains the user-defined chains.

The mapping function $c\colon Q \mapsto \rho$ is defined as follows:

$$c(q_i) = c(q_f) = R \qquad c(q_0) = c(q_{12}) = R^{man}_{\text{PRE}} \qquad c(q_1) = R^{nat}_{\text{PRE}}$$
$$c(q_2) = R^{man}_{\text{FOR}} \qquad c(q_3) = R^{fil}_{\text{FOR}} \qquad c(q_4) = c(q_{13}) = R^{man}_{\text{INP}}$$

$$c(q_5) = R^{nat}_{\text{INP}} \qquad c(q_6) = R^{fil}_{\text{INP}} \qquad c(q_7) = R^{man}_{\text{OUT}}$$
$$c(q_8) = R^{nat}_{\text{OUT}} \qquad c(q_9) = R^{fil}_{\text{OUT}} \qquad c(q_{10}) = R^{man}_{\text{POST}}$$
$$c(q_{11}) = R^{nat}_{\text{POST}}$$

where R is an empty ruleset with ACCEPT as default policy.

Finally, note that the IFCL action CALL(_) implements the `iptables` built in target JUMP(_).

pf A `pf` configuration is basically a single list of rules and the rule applied to a packet is the last matching one, except in the case of the so-called `quick` rules: as soon as one of these matches the packet, its action is applied and the remaining part of the ruleset is skipped.

Figure 2.2b shows the control diagram $\mathcal{C}_{\texttt{pf}}$ for `pf`. The nodes q_0 and q_1 represent the procedure executed when an IP packet leaves the host. Dually, q_2 and q_3 are for when the packet reaches the host from the net. Given the `pf` ruleset $R_{\texttt{pf}}$ we include the following rulesets in ρ_{pf}:

- R_{dnat} contains the rule ($state == 1$, CHECK-STATE($\rightarrow$)) as the first one, followed by all the rules `rdr` of $R_{\texttt{pf}}$;
- R_{snat} contains the rule ($state == 1$, CHECK-STATE($\leftarrow$)) as the first one, followed by all the rules `nat` of $R_{\texttt{pf}}$;
- R_{finp} contains the rule ($state == 1$, ACCEPT) followed by all the `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `out`, and finally the rule ($true$, GOTO(R_{finpr}));
- R_{finpr} contains all the no `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `out`, in reverse order;
- R_{fout} contains the rule ($state == 1$, ACCEPT) followed by all the `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `in`, and ($true$, GOTO(R_{foutr})) as last rule;
- R_{foutr} includes all the no `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `in` in reverse order.

Given the empty ruleset R with ACCEPT as default policy, the mapping function c_{pf} is:

$$c_{pf}(q_i) = R \qquad c_{pf}(q_0) = R_{snat} \qquad c_{pf}(q_2) = R_{dnat}$$
$$c_{pf}(q_f) = R \qquad c_{pf}(q_1) = R_{fout} \qquad c_{pf}(q_3) = R_{finp}$$

ipfw Also `ipfw` has a single ruleset and, differently from above, the rule applied to a packet is the first matching one.

The control diagram $\mathcal{C}_{\texttt{ipfw}}$ of `ipfw`, displayed in Figure 2.2c. The node q_0 represents the procedure executed when an IP packet reaches the host from the net. Dually, q_1 is for when the packet leaves the host.

We present the construction of the rulesets associated with the node q_0. Let $R = [r_{id_1}, \ldots, r_{id_k}]$ be the unique ruleset of `ipfw`, where id_i are numeric

identifiers associated with the rules, and the last rule r_{id_k} encodes the default policy set by the user. The idea is to generate k different rulesets R_i^I, one for each rule in R. If the rule r_{id_i} contains the keyword `out`, i.e., the rule is not considered when the packet enters the firewall, we let $R_i^I = [(true, \texttt{GOTO}(R_{i+1}^I))]$. Otherwise, we define $R_i^I = [trs(r_{id_i}), (true, \texttt{GOTO}(R_{i+1}^I))]$, where the translation trs is defined by cases below:

$$trs(r) = \begin{cases} (\phi, \texttt{GOTO}(R_n^I)) & \text{if } r = (\phi, \texttt{skipto } id_n) \\ (\phi, \texttt{CALL}(R_n^I)) & \text{if } r = (\phi, \texttt{call } id_n) \\ (\phi, \texttt{t}) & \text{if } r = (\phi, \texttt{t}) \end{cases}$$

The construction of the rulesets R_i^O for the node q_1 is similar, but in this case the rules containing the keyword `in` are ignored. The mapping function c returns R_1^I for q_0, R_1^O for q_1, and empty ruleset with ACCEPT as default policy for q_i and q_f. These rulesets form the component $\rho_{\texttt{ipfw}}$.

2.2.3 Legal Firewalls

Not every target can be inside the rulesets associated with nodes in the control diagram of a firewall. Because of this, we require that each node q of $C_{\mathcal{L}}$ also carries information on which operations can be applied to packets when in node q. For example, in the encoding of `pf`, the rules `rdr` are only assigned to the node q_2. Hence, a correspondence mapping associating the node q_2 with a ruleset where a target DROP appears is not valid for `pf`, yet it is for IFCL.

We represent this additional information in a handy manner by decorating each node q with *cap-labels* representing the target allowed in q. Note that this constrain also the rulesets called by the ruleset associated with q. We only focus on ACCEPT, DROP and NAT, since marking, control-flow and state related actions are considered in ad hoc ways.

Formally, we are making explicit that languages put constraints by restricting the image of the function c. Figure 2.2 shows the cap-labels within rectangles (we assume ID, representing the ACCEPT target, be always present, therefore omitted).

Definition 2.7 (Cap-labels and allowed actions)**.** Given a control diagram $\mathcal{C}$ with nodes Q, and the set of cap-labels $\mathbb{L} = \{\texttt{ID}, \texttt{DNAT}, \texttt{SNAT}, \texttt{DROP}\}$, a *cap-label assignment* is a function $V \colon Q \to 2^{\mathbb{L}}$.
Furthermore, we define the mapping *actions* that associates a cap-label $l \in \mathbb{L}$ with the set of allowed actions as follows:

$$actions(\texttt{ID}) = \{\texttt{ACCEPT}\} \qquad actions(\texttt{DROP}) = \{\texttt{DROP}\}$$
$$actions(\texttt{DNAT}) = \{\texttt{NAT}(n_d, \star)\} \qquad actions(\texttt{SNAT}) = \{\texttt{NAT}(\star, n_s)\}$$

We extend *actions* to homomorphically operate on sets $L \subseteq \mathbb{L}$ of labels, i.e. returning the union of $actions(l)$, for $l \in L$.

We call *legal* those firewalls that respect the cap-label assignment. Formally,

Definition 2.8 (Legal firewall)**.** A firewall $(\mathcal{C}, \rho, c)$ with nodes Q is *legal for a cap-label assignment* V if and only if for each node $q \in Q$, $\mathsf{Actions}(V(q))$ includes all the actions appearing in $c(q)$ and in the called rulesets, apart from `MARK`, `CHECK-STATE`, `CALL`, `GOTO` and `RETURN`.

Given a language $\mathcal{L}$, let hereafter $V_{\mathcal{L}}$ be the cap-label assignment for $\mathcal{L}$.

2.2.4 Operational Semantics

Now, we formally define the operational semantics of IFCL through two transition systems operating in a master-slave fashion. The master has a labeled transition relation of the form $s \xrightarrow{p,p'} s'$. The intuition is that the state s of a firewall changes to s' when a new packet p reaches the host and becomes p'.

The configurations of the slave transition system are triples (q, s, p) where:

(i) $q \in Q$ is a control diagram node; (ii) s is the state of the firewall; (iii) p is the packet.

A transition $(q, s, p) \to (q', s, p')$ describes how a firewall in a state s deals with a packet p and possibly transforms it in p', according to the control diagram $\mathcal{C}$. Recall that the state records established connections and other kinds of information that are updated after the transition.

In the slave transition relation, we use the following predicate, which describes an algorithm that runs a ruleset R on a packet p in the state s

$$p, s \models^{S}_{R} (t, p')$$

This predicate searches for a rule in R matching the packet p through $p \Vdash_R (t, i)$. If it finds a match with target t, t is applied to p to obtain the packet p' resulting from possible transformations.

Recall that actions `CALL(R)`, `RETURN` and `GOTO(R)` are similar to procedure calls, returns and jumps in imperative programming languages. To correctly deal with them, our predicate $p, s \models^{S}_{R} (t, p')$ uses a stack S to implement a behavior similar to the one of procedure calls. We will denote with ϵ the empty stack and with $\cdot$ the concatenation of elements on the stack. This stack is also used to detect and prevent loops in ruleset invocation, as it is the case in real firewalls.

In the stack S we overline a ruleset R to indicate that it was pushed by a `GOTO(_)` and it has to be skipped when returning. For that, we use the following pop^* function in the semantics of the `RETURN`:

$$pop^*(\epsilon) = \epsilon \qquad pop^*(R \cdot S) = (R, S) \qquad pop^*(\overline{R} \cdot S) = pop^*(S)$$

If the top of S is overlined, pop^* behaves as a standard pop operation; otherwise it extracts the first non-overlined ruleset. When S is empty, we assume that pop^* returns ϵ to signal the error.

Furthermore, in the definition of $p, s \models^{S}_{R} (t, p')$ the notation R_k indicates the ruleset $[r_k, ..., r_n]$ $(k \in [1, n])$ resulting from dropping the first $k - 1$ rules from the given ruleset $R = [r_1, ..., r_n]$.

Table 2.1: The predicate $p, s \models^S_R (t, p')$.

$$(1)\ \frac{p \Vdash_R (t, i) \quad t \in \{\texttt{ACCEPT}, \texttt{DROP}\}}{p, s \models^S_R (t, p)} \qquad (2)\ \frac{p \Vdash_R (\texttt{CHECK-STATE}(X), i) \quad p \vdash_s \alpha \quad p' = \mathit{establ}(\alpha, X, p)}{p, s \models^S_R (\texttt{ACCEPT}, p')}$$

$$(3)\ \frac{p \Vdash_R (\texttt{CHECK-STATE}(X), i) \quad p \nvdash_s \quad p, s \models^S_{R_{i+1}} (t, p')}{p, s \models^S_R (t, p')} \qquad (4)\ \frac{p \Vdash_R (\texttt{NAT}(n_d, n_s), i)}{p, s \models^S_R (\texttt{ACCEPT}, \mathit{nat}(p, s, n_d, n_s))}$$

$$(5)\ \frac{p \Vdash_R (\texttt{GOTO(R')}, i) \quad R' \notin S \quad p, s \models^{\overline{R}, S}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (6)\ \frac{p \Vdash_R (\texttt{GOTO(R')}, i) \quad R' \in S}{p, s \models^S_R (\texttt{DROP}, p)}$$

$$(7)\ \frac{p \Vdash_R (\texttt{CALL(R')}, i) \quad R' \notin S \quad p, s \models^{R_{i+1} \cdot S}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (8)\ \frac{p \Vdash_R (\texttt{CALL(R')}, i) \quad R' \in S}{p, s \models^S_R (\texttt{DROP}, p)}$$

$$(9)\ \frac{p \Vdash_R (\texttt{RETURN}, i) \quad \mathit{pop}^*(S) = (R', S') \quad p, s \models^{S'}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (10)\ \frac{p \Vdash_R (\texttt{RETURN}, i) \quad \mathit{pop}^*(S) = \epsilon}{p, s \models^S_R (t_d, p)}$$

$$(11)\ \frac{p \nVdash_R \quad S \neq \epsilon \quad \mathit{pop}^*(S) = (R', S') \quad p, s \models^{S'}_{R'} (t, p')}{p, s \models^S_R (t, p')} \qquad (12)\ \frac{p \nVdash_R \quad (S = \epsilon \ \vee \ \mathit{pop}^*(S) = \epsilon)}{p, s \models^S_R (t_d, p)}$$

$$(13)\ \frac{p \Vdash_R (\texttt{MARK}(m), i) \qquad p[\mathit{tag} \mapsto m], s \models^S_{R_{i+1}} (t, p')}{p, s \models^S_R (t, p')}$$

We also assume the function *establ* that, taken an action α from the state, a packet p and the fields $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ to rewrite, returns a possibly changed packet p', e.g., in case of an established connection. Also *establ* depends on the specific firewall we are modeling, and so it is left unspecified.

Finally, we assume as given a function $\mathit{nat}(p, s, n_d, n_s)$ that returns the packet p translated under the corresponding NAT operation in the state s. The argument n_d is used to modify the destination range of p, i.e., destination NAT (DNAT), while n_s is used to modify the source range, i.e., source NAT (DNAT). Recall that a range of the form $\star : \star$ is interpreted as the identity translation, whereas one of the form $a : \star$ modifies only the destination address. Also this function is left abstract.

Table 2.1 shows the rules defining $p, s \models^S_R (t, p')$. The first inference rule deals with the case when the packet p matches a rule with target ACCEPT or DROP; in this case the ruleset execution stops returning the found action and leaving p unmodified. When a packet p matches a rule with action CHECK-STATE, we query the state s: if p belongs to an established connection, we return ACCEPT and p', obtained rewriting p. Otherwise, p is matched against the remaining rules in the ruleset. When a packet matches a NAT rule, we return ACCEPT and the packet resulting by the invocation of the function *nat*. There are two cases when a packet p matches a GOTO(_). If the ruleset R' is not already in the stack, we push

the current ruleset R onto the stack overlined to record that this ruleset dictated a GOTO(_). Otherwise, R' is in the stack, we detect the loop and discard p. The case when a packet p matches a rule with action CALL(_) is similar, except that the ruleset pushed on the stack is not overlined. When p matches a rule with action RETURN, we pop the stack and match p against the top of the stack. Finally, when no rule matches, an implicit return occurs: we continue from the top of the stack, if non empty. The MARK rule simply changes the tag of the matching packet to the value m. If none of the above applies, we return the default action t_d of the current ruleset.

We now define the slave transition relation as follows.

$$\frac{c(q) = R \quad p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p') \quad \delta(q, p') = q'}{(q, s, p) \to (q', s, p')}$$

The rule describes how we process the packet p when the firewall is in state s and performs the step represented by the node q. We match p against the ruleset R associated with q and if p is accepted as p', we continue considering the next step of the firewall execution represented by the node q'.

Finally, we define the master transition relation that transforms states and packets as follows (as usual, below $\to^+$ stands for the transitive closure of $\to$):

$$\frac{(q_i, s, p) \to^+ (q_f, s, p')}{s \xrightarrow{p,p'} s \uplus (p, p')}$$

This rule says that when the firewall is in the state s and receives a packet p, it elaborates p starting from the initial node q_i of its control diagram. If this elaboration succeeds, i.e., if the node q_f is reached, p is accepted as p', and we update the state s by storing information about p, its translation p' and the connection they belong to, by the function $\uplus$, left unspecified for the sake of generality.

Example 2.1. Suppose to have the rulesets below:

Ruleset C_B	Ruleset u_1	Ruleset u_2
$(\phi_1, \texttt{DROP})$	$(\phi_{11}, \texttt{ACCEPT})$	$(\phi_{21}, \texttt{ACCEPT})$
$(\phi_2, \texttt{CALL}(u_1))$	$(\phi_{12}, \texttt{CALL}(u_2))$	$(\phi_{22}, \texttt{RETURN})$
$(\phi_3, \texttt{ACCEPT})$	$(\phi_{13}, \texttt{DROP})$	$(\phi_{23}, \texttt{DROP})$

and that the condition $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ holds for a packet p. Then, the semantic rules (a), (b) and (c) are applied in order:

$$(a)\ \frac{p \models_{u_1} (\texttt{ACCEPT}, 1)}{p, s \models^{C_{B_3} \cdot \epsilon}_{u_1} (\texttt{ACCEPT}, p)} \qquad (b)\ \frac{p \models_{C_B} (\texttt{CALL}(u_1), i) \quad u_1 \notin S \quad (a)}{p, s \models^{\epsilon}_{C_B} (\texttt{ACCEPT}, p)}$$

$$(c)\ \frac{c(q) = C_B \quad (b) \quad \delta(q, p) = q'}{(q, s, p) \to (q', s, p)}$$

Semantics validation We empirically validated the semantics inherited from IFCL by the firewall systems we have considered in Section 2.1. The validation process is composed of three main phases.

We first inferred the expected behavior of the various processing steps of the real firewall system $\mathcal{F}$ in hand from its documentation, and collected a set of configurations playing the role of test cases T. We then run $\mathcal{F}$ in a simulated environment, built on some virtual machines. Also, we synthesize the abstract specification of $\mathcal{F}$ through FWS. Furthermore, we generated synthetic network traffic to make sure that we cover all the test cases in T. Finally, we compare the logs of $\mathcal{F}$ against the results obtained by FWS, so as to check the correspondence between the actual behavior of $\mathcal{F}$ and the one dictated by the semantics it inherits from IFCL.

We focused on corner cases that are not clearly explained in the official documentation of $\mathcal{F}$, or not even taken into account. Feedback from system administrators has been particularly helpful in detecting these cases. Often the corner cases concern local packets, i.e., packets generated by the firewall and directed to a local address of the firewall itself, in combination with NAT rules.

As an example, consider `iptables`. A local packet first enters the `Output` chain, being generated by the host, and then the `PostRouting` chain. Instead of leaving the firewall, it enters the `PreRouting` and `Input` chains, being also a packet directed to a local address. In these two chains, the processing of this kind of packets differs from the one of a generic packet coming from outside $\mathcal{F}$. Indeed a local packet does not traverse the `nat` table in the `PreRouting` and `Input` chains, skipping the NAT processing. This behavior is correctly encoded in the control diagram of Figure 2.2a. The validation phase sketched above helped us in discovering this subtle behavior, which was not considered in the preliminary version of the `iptables` control diagram [27].

2.2.5 Normal form

For convenience, we reduce configurations to a normal form that will simplify the compilation and decompilation, as well as the study of the expressivity of the different configuration languages. A configuration is in normal form if it does not contain any control flow instruction, i.e., CALL(), GOTO(), and RETURN, used by our intermediate language for dealing with involved control flows (see Example 2.1).

The following unfolding operation $(\!|\ \_\ |\!)$ rewrites a ruleset into an equivalent one with no such control flow rules.

Hereafter, let $r;R$ be a non empty ruleset consisting of a rule r followed by a possibly empty ruleset R; and let $R_1@R_2$ be the concatenation of R_1 and R_2.

The unfolding of a ruleset R is defined as follows:

$$
\begin{aligned}
(\!|\ R\ |\!) &= (\!|\ R\ |\!)^{true}_{\{R\}}\bullet \\
(\!|\ \epsilon\ |\!)^f_I m &= \epsilon \\
(\!|\ (\phi,t);R\ |\!)^f_I m &= (f\wedge\phi',t);(\!|\ R\ |\!)^f_I m \quad \text{if } t\notin\{\texttt{RETURN},\texttt{CALL(R')},\texttt{GOTO(R')},\texttt{MARK}(m)\} \\
(\!|\ (\phi,\texttt{RETURN});R\ |\!)^f_I m &= (\!|\ R\ |\!)^{f\wedge\neg\phi'}_I m \\
(\!|\ (\phi,\texttt{CALL(R')});R\ |\!)^f_I m &= \begin{cases} (\!|\ R'\ |\!)^{f\wedge\phi'}_{I\cup\{R'\}} m@(\!|\ R\ |\!)^f_I m & \text{if } R'\notin I \\ (f\wedge\phi',\texttt{DROP});(\!|\ R\ |\!)^f_I m & \text{otherwise} \end{cases}
\end{aligned}
$$

$$\llparenthesis (\phi, \texttt{GOTO(R')}); R \rrparenthesis_I^f m = \begin{cases} \llparenthesis R' \rrparenthesis_{I \cup \{R'\}}^{f \wedge \phi'} m @ \llparenthesis R \rrparenthesis_I^{f \wedge \neg \phi'} m & \text{if } R' \notin I \\ (f \wedge \phi', \texttt{DROP}); \llparenthesis R \rrparenthesis_I^{f \wedge \neg \phi'} m & \text{otherwise} \end{cases}$$

$$\llparenthesis (\phi, \texttt{MARK}(m')); R \rrparenthesis_I^f m = (f \wedge \phi', \texttt{MARK}(m')); \llparenthesis R \rrparenthesis_I^{f \wedge \phi'} m' @ \llparenthesis R \rrparenthesis_I^{f \wedge \neg \phi'} m$$

where ϕ' is ϕ with its sub-predicate on the current tag replaced by its evaluation on m.

The auxiliary procedure $\llparenthesis R \rrparenthesis_I^f m$ recursively inspects the ruleset R. The formula f accumulates conjuncts of the predicate ϕ'; the set I stores the rulesets traversed by the procedure and helps detecting loops; m records the tag associated with the packets that satisfy f. If a rule neither affects the control flow nor it is a MARK, we just substitute the conjunction $f \wedge \phi'$ for ϕ, and continue to analyze the rest of the ruleset with the recursive call $\llparenthesis R \rrparenthesis_I^f m$.

In the case of a return rule $(\phi, \texttt{RETURN})$ we generate no new rule, and we continue to recursively analyze the rest of the ruleset, by updating f with the negation of ϕ'. For the rule $(\phi, \texttt{CALL(R')})$ we have two cases: if the callee ruleset R' is not in I, we replace the rule with the unfolding of R' with $f \wedge \phi'$ as predicate, and append $\{R'\}$ to the traversed rulesets; if R' is already in I, i.e., we have a loop, we replace the rule with a DROP, with $f \wedge \phi'$ as predicate; in both cases, we continue unfolding the rest of the ruleset. We deal with the rule $(\phi, \texttt{GOTO(R')})$ as in the previous one, except that the rest of the ruleset has $f \wedge \neg \phi'$ as predicate. Finally, the rule $(\phi, \texttt{MARK}(m'))$ originates two lists. The first is for the packets satisfying ϕ' and thus the rule becomes $(f \wedge \phi', \texttt{MARK}(m'))$ and its tag m'. The other list is for packets not satisfying ϕ', so $\texttt{MARK}(m')$ never applies and the tag m is left unchanged.

Example 2.2. Back to Example 2.1, unfolding the ruleset C_B gives the following rules:

$$\begin{aligned} \llparenthesis C_B \rrparenthesis = &(\phi_1, \texttt{DROP}); (\phi_2 \wedge \phi_{11}, \texttt{ACCEPT}); (\phi_2 \wedge \phi_{12} \wedge \phi_{21}, \texttt{ACCEPT}); (\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}, \texttt{DROP}); \\ &(\phi_2 \wedge \phi_{13}, \texttt{DROP}); (\phi_3, \texttt{ACCEPT}); \epsilon \end{aligned}$$

We just illustrate the first three steps:

$$\begin{aligned} \llparenthesis C_B \rrparenthesis &= \llparenthesis (\phi_1, \texttt{DROP}); C_{B2} \rrparenthesis_{\{C_B\}}^{true} \bullet \\ &= (\phi_1, \texttt{DROP}); \llparenthesis (\phi_2, \texttt{CALL}(u_1)); C_{B3} \rrparenthesis_{\{C_B\}}^{true} \bullet \\ &= (\phi_1, \texttt{DROP}); \llparenthesis u_1 \rrparenthesis_{\{C_B\} \cup \{u_1\}}^{true \wedge \phi_2} \bullet @ \llparenthesis C_{B3} \rrparenthesis_{\{C_B\}}^{true} \bullet \end{aligned}$$

Note that the packet p is accepted also by the unfolded firewall, provided that $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ holds.

An unfolded firewall is obtained by repeatedly rewriting the rulesets associated with the nodes of its control diagram, using the procedure above. Formally:

Definition 2.9 (Unfolded firewall)**.** Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$, its unfolded version $\llparenthesis \mathcal{F} \rrparenthesis$ is $(\mathcal{C}, \rho', c')$ where $\rho' = \{\llparenthesis c(q) \rrparenthesis \mid q \in \mathcal{C}\}$, $\forall q \in Q.\, c'(q) = \llparenthesis c(q) \rrparenthesis$.

We now prove that a firewall $\mathcal{F}$ and its unfolded version $\llparenthesis \mathcal{F} \rrparenthesis$ are semantically equivalent, i.e., they perform the same action over a given packet p in a state s, and reach the same state s'. Formally:

Theorem 2.1 (Correctness of unfolding)**.** Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall and $(\!|\mathcal{F}|\!)$ its unfolding. Let $s \xrightarrow{p,p'}_X s'$ be a step of the master transition system performed by the firewall $X \in \{\mathcal{F}, (\!|\mathcal{F}|\!)\}$. Then

$$s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{(\!|\mathcal{F}|\!)} s'.$$

2.3 Modeling the High Level

Nowadays, network administrators exploit policy-based management systems and work with high-level and user-friendly policy languages that are then often compiled to configurations of a target system (e.g., iptables on Linux). Following this line, we propose FWQL, a firewall language that allows the administrator to interact with a firewall. The policy implemented by the firewall is represented as a function, visualized as a table, listing the accepted packets and their transformations in a succinct way. The administrator can thus interact with the policy by investigating the treatment of specific packets to check the correctness of their management, or update the table.

2.3.1 Firewalls as functions

A firewall either leaves a packet unchanged, or it modifies some of its fields. We formalize this activity by the transformation functions associated with the nodes of the control diagram. A *transformation t of a field w* is either the identity function id (the value of w is left unchanged) or the constant function $\lambda_{a'}$ returning the value $a' \in D_w$ (the value of w is now a').

A *packet transformation* $t = (t_{dIP} : t_{dPort}, t_{sIP} : t_{sPort}) \in \mathcal{T}_{\mathbb{P}}$ is a quadruple of transformations, two for the IP and port of the destination fields, and two for the source fields. Packet transformations are applied and composed componentwise.

For convenience, we extend the set of packets $\mathbb{P}$ with the distinguished element $\perp$ to represent the dropped packets. Thus, transformations are blankly extended, assuming $t(\perp) = \perp$. Also, we denote with $\lambda_\perp$ the transformation that always drops packets.

Example 2.3. Let the packet p have $da(p) = 8.8.8.8 : 53$ and $sa(p) = 192.168.0.8 : 50000$, and let $t = (id : id, \lambda_{151.15.1.5} : id)$ be the transformation that changes the source IP (performing a `SNAT`); then $p' = t(p)$ has $da(p') = 8.8.8.8 : 53$ and $sa(p') = 151.15.1.5 : 50000$.

In the high layer, a firewall is a *fw-function*, i.e. a function $\tau : \mathbb{P} \to \mathcal{T}_{\mathbb{P}}$.

2.3.2 Effective Representation of Firewalls

In set theoretical terms, a fw-function τ, is a set of pairs (p, t). To concisely represent these pairs, we first group in a set P all the packets subject to the same transformation t, and we present the function τ as a set of *τ-pairs* (P, t).

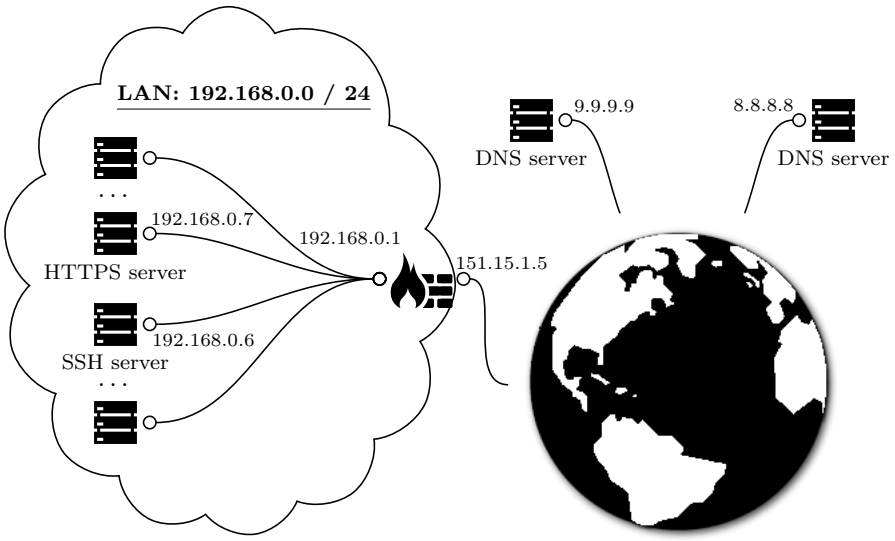


Figure 2.3: A network of example.

We then efficiently represent fw-functions using multi-cubes [78]. A multi-cube generalizes the notion of cube, and can be considered as the cartesian product of the union of intervals. A multi-cube compactly represents a set of packets, and each union specifies the interval to which the values of the corresponding field of a packet belong.

Since a set P in a τ-pair not always forms a multi-cube, we need an intermediate step. In such a case, P is partitioned in a set of P_i each originating a multi-cube.

A fw-function can be efficiently represented as a table. We will usually write only the rows of accepted packets, assuming that the omitted ones are mapped to $\lambda_\perp$.

Example 2.4. Consider the following multi-cube:

$$([192.198.0.2, 192.198.0.10] : [8080, 8080],$$
$$[192.198.0.1, 192.198.0.1] : [0, 22] \cup [25, 100])$$

It represents the set of packets with destination address in $[192.198.0.2, 192.198.0.10]$ and port 8080, and source address 192.198.0.1 with source port in the intervals $[0, 22]$ or $[25, 100]$.

2.3.3 The FireWall Query Language FWQL

Since a firewall is essentially represented as a table, a SQL-like query language can be used to verify properties, extract and visualize portions of the function,

create or update it. We will overview FWQL features with examples.

Consider the network in Figure 2.3. The firewall at the addresses S = 192.168.0.1, 151.15.1.5 is the only connection point between the Internet and a Local Area Network (LAN). The LAN private addresses range over 192.168.0.0/24; the internal hosts at 192.168.0.6 and 192.168.0.7 run an SSH and an HTTPS server. On the Internet, two DNS servers are hosted at 8.8.8.8 and 9.9.9.9.

Assume that the fw-function of the firewall in hand is named `TAB`. The following can be used to verify that every packet coming from a LAN host or the firewall itself, and directed to 8.8.8.8 are redirected to the host at 9.9.9.9.

```
verify in TAB where
  ( (srcIp = 192.168.0.0/24 or srcIp = 151.15.1.5)
  and dstIp = 8.8.8.8 )
that t_dst = 9.9.9.9
```

Assuming the previous command returns *false*, the user can look at the subtable concerning only the packets of interest, with their associated transformations with the following query.

```
print * in TAB where
( (srcIp = 192.168.0.0/24 or srcIp = 151.15.1.5)
and dstIp = 8.8.8.8 )
```

Finally, the following command updates the transformation with the desired one.

```
update t_dst = 9.9.9.9 in TAB where
( (srcIp = 192.168.0.0/24 or srcIp = 151.15.1.5)
and dstIp = 8.8.8.8 )
```

2.4 Decompilation

The decompilation from configuration to table is obtained through a pipeline of stages

1. encoding of the configuration from the source language to IFCL;
2. normalization of the IFCL configuration;
3. generation of the table.

The encoding of stage 1 and the normalization follows the procedures described in subsection 2.2.2 and subsection 2.2.5, respectively. The generation of the table representing the semantics of a normalized firewall configuration written in IFCL is obtained by first obtaining a declarative, logical characterization of the configuration in hand. The rows of the table contains the models of the logical formulas. Of course this transformation is independent of the firewall language in hand.

The correctness of stage 2 follows from Theorem 2.1, which guarantees that the unfolded firewall is semantically equivalent to the original one; and the correctness of stage 3 follows from Theorem 2.2, which ensures that the derived

Table 2.2: Translation of rulesets into the logical predicates P_R for accepted and D_R for dropped packets

.

$$
\begin{aligned}
P_\epsilon(p,\tilde{p}) &= dp(R) \wedge p = \tilde{p} \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge p = \tilde{p}) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{ACCEPT}) \\
P_{r;R}(p,\tilde{p}) &= \neg\phi(p) \wedge P_R(p,\tilde{p}) && \text{if } r = (\phi, \texttt{DROP}) \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge \tilde{p} \in tr(p, n_d, n_s, \leftrightarrow)) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{NAT}(n_d, n_s)) \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge \tilde{p} \in tr(p, *{:}*, *{:}*, X) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{CHECK-STATE}(X)) \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge P_R(p[tag \mapsto m],\tilde{p})) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{MARK}(m))
\end{aligned}
$$

$$
\begin{aligned}
D_\epsilon(p) &= \neg dp(R) \\
D_{r;R}(p) &= \neg\phi(p) \wedge D_R(p) && \text{if } r = (\phi, \texttt{ACCEPT}) \\
D_{r;R}(p) &= \phi(p) \vee (\neg\phi(p) \wedge D_R(p)) && \text{if } r = (\phi, \texttt{DROP}) \\
D_{r;R}(p) &= \neg\phi(p) \wedge D_R(p) && \text{if } r = (\phi, \texttt{NAT}(n_d, n_s)) \\
D_{r;R}(p) &= \neg\phi(p) \wedge D_R(p) && \text{if } r = (\phi, \texttt{CHECK-STATE}(X)) \\
D_{r;R}(p) &= (\phi(p) \wedge D_R(p[tag \mapsto m])) \vee (\neg\phi(p) \wedge D_R(p)) && \text{if } r = (\phi, \texttt{MARK}(m))
\end{aligned}
$$

formula characterizes exactly the accepted/dropped packets and their translations.

2.4.1 Logical characterization of firewalls

We construct two logical predicates that characterize the packets accepted and those dropped by an unfolded ruleset, together with the relevant translations.

To deal with NAT, we define an auxiliary function tr that computes the set of packets resulting from all possible translations of a given packet p. The parameter $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ specifies if the translation applies to source, destination or both addresses, respectively, similarly to $\texttt{CHECK-STATE}(X)$.

$$
\begin{aligned}
tr(p, n_d, n_s, \leftrightarrow) &\triangleq \{p[da \mapsto a_d, sa \mapsto a_s] \mid a_d \in n_d, a_s \in n_s\} \\
tr(p, n_d, n_s, \rightarrow) &\triangleq \{p[da \mapsto a_d] \mid a_d \in n_d\} \\
tr(p, n_d, n_s, \leftarrow) &\triangleq \{p[sa \mapsto a_s] \mid a_s \in n_s\}
\end{aligned}
$$

Furthermore, we model the default policy of a ruleset R with the predicate dp, *true* when the policy is ACCEPT, *false* otherwise.

Given an unfolded ruleset R, we build two predicates: $P_R(p,\tilde{p})$ that holds when the packet p is accepted as $\tilde{p}$ by R, and $D_R(p)$ when p is instead dropped. Their definition in Table 2.2 induces on the rules in R. Recall that formulas ϕ also predicate on the firewall state, e.g., checking if the packet belongs to an established connection. As a sanity check, we assume that this is always the case when the target is CHECK-STATE. We briefly comment on the definition of $P_R(p,\tilde{p})$.

The empty ruleset applies the default policy $dp(R)$ and does not transform the packet, reflected by the constraint $p = \tilde{p}$. The rule $(\phi, \texttt{ACCEPT})$ considers two cases: when $\phi(p)$ holds and the packet is accepted as it is; when instead $\neg\phi(p)$ holds, p is accepted as $\tilde{p}$ only if the continuation R accepts it. The rule $(\phi, \texttt{DROP})$ accepts p only if the continuation does and $\phi(p)$ is false. The rule $(\phi, \texttt{NAT}(n_d, n_s))$ works similarly to the rule $(\phi, \texttt{ACCEPT})$, expect that, when $\phi(p)$ holds, and it gives $\tilde{p}$ by applying to p the NAT translations $tr(p, n_d, n_s, \leftrightarrow)$. Finally, $(\phi, \texttt{CHECK-STATE}(X))$ works similarly to the rule a NAT that applies all possible translations of kind X (written as $tr(p, *{:}*, *{:}*, X)$). Intuitively, we over-approximate the state by considering any possible translations, because we abstract away from the actual established connections. At run-time, only the connections corresponding to the actual state will be possible. The rule $(\phi, \texttt{MARK}(m))$ works similarly to the rule NAT, but when $\phi(p)$ holds it requires that the continuation accepts p tagged by m as $\tilde{p}$. Apart from being monadic, the definition of $D_R(p)$ follows the same schema.

Example 2.5. The predicates P_R and D_R of the ruleset in Example 2.2 when $dp(C_B) = \mathit{false}$ is

$$
\begin{aligned}
&P_{(\!| C_B |\!)}(p, \tilde{p}) = \\
&\quad \neg\phi_1 \wedge (\\
&\quad (\phi_2 \wedge \phi_{11} \wedge p = \tilde{p}) \vee (\neg(\phi_2 \wedge \phi_{11}) \wedge (\\
&\quad (\phi_2 \wedge \phi_{12} \wedge \phi_{21} \wedge p = \tilde{p}) \vee (\neg(\phi_2 \wedge \phi_{12} \wedge \phi_{21}) \wedge (\\
&\quad \neg(\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}) \wedge (\\
&\quad \neg(\phi_2 \wedge \phi_{13}) \wedge (\\
&\quad (\phi_3 \wedge p = \tilde{p}) \vee (\neg\phi_3 \wedge (\\
&\quad \mathit{false} \wedge p = \tilde{p}))))))))
\end{aligned}
$$

$$
\begin{aligned}
&D_{(\!| C_B |\!)}(p) = \\
&\quad \phi_1 \vee (\neg\phi_1 \wedge (\\
&\quad \neg(\phi_2 \wedge \phi_{11}) \wedge (\\
&\quad \neg(\phi_2 \wedge \phi_{12} \wedge \phi_{21}) \wedge (\\
&\quad (\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}) \vee (\\
&\quad \neg(\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}) \wedge (\\
&\quad (\phi_2 \wedge \phi_{13}) \vee (\neg(\phi_2 \wedge \phi_{13}) \wedge (\\
&\quad \neg\phi_3 \wedge \mathit{true}))))))))
\end{aligned}
$$

Note that if $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ holds then the formula trivially holds and therefore the formula accepts the packet as the semantics does.

Also, consider the case in which $\phi_2, \phi_{12}, \phi_{22}, \phi_{23}, \phi_3$ hold for a packet p, while all the other predicates do not. Then, p is accepted as it is: the rule $(\phi_{23}, \texttt{DROP})$ is not evaluated since ϕ_{22} holds and the RETURN is performed (cf. Example 2.1). Indeed, the predicate $P_{(\!| C_B |\!)}(p, p)$ evaluates to the following, where we abbreviate true with T and false with F:

$$T \wedge (F \vee (T \wedge (F \vee (T \wedge (T \wedge (T \wedge (T \vee (F \wedge F)))))))) = T$$

Instead, if ϕ_{13} holds too, the packet is rejected as expected:

$$T \wedge (F \vee (T \wedge (F \vee (T \wedge (T \wedge (F \wedge (T \vee (F \wedge F)))))))) = F$$

Example 2.6. Let $(sa(p) = 1 : 22, \texttt{NAT}(\{2, 3\} : 22, \star : \star)); (da(p) = 2 : 22, \texttt{DROP})$ be the ruleset R, where NAT maps to a set of IP addresses. Now, $(p, \tilde{p}) \in P_R$ where $da(p)$ is any, $sa(p) = sa(\tilde{p}) = 1 : 22$, and $da(\tilde{p}) = 2 : 22$. Therefore, R accepts p as $\tilde{p}$, but it drops $\tilde{p}$ itself, because it belongs to D_R.

The predicate $P_R(p, p')$ in Table 2.2 is semantically correct, because if a packet p is accepted by a ruleset R as p', then $P_R(p, p')$ holds, and vice versa. Formally:

Lemma 2.1. Given a ruleset R we have that

1. $\forall p, s.\ p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p') \implies P_R(p, p')$; and
2. $\forall p, p'.\ P_R(p, p') \implies \exists s.\ p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p')$

We eventually define the predicates associated with a whole firewall as follows.

Definition 2.10. Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall with control diagram $\mathcal{C} = (Q, A, q_i, q_f)$. The predicates $\mathcal{P}_{\mathcal{F}}(p, \tilde{p})$ and $\mathcal{D}_{\mathcal{F}}(p)$ associated with $\mathcal{F}$ are defined as follows:

$$\mathcal{P}_{\mathcal{F}}(p, \tilde{p}) \triangleq \mathcal{P}^{\emptyset}_{q_i}(p, \tilde{p}) \quad \text{and} \quad \mathcal{D}_{\mathcal{F}}(p) \triangleq \mathcal{D}^{\emptyset}_{q_i}(p) \qquad \text{where}$$

$$\mathcal{P}^{I}_{q_f}(p, \tilde{p}) \triangleq p = \tilde{p} \qquad \mathcal{P}^{I}_{q}(p, \tilde{p}) \triangleq \exists p'.P_{c(q)}(p, p') \wedge \left(\bigvee_{\substack{(q,\psi,q') \in A \\ q' \notin I}} \psi(p') \wedge \mathcal{P}^{I \cup \{q\}}_{q'}(p', \tilde{p}) \right)$$

$$\mathcal{D}^{I}_{q_f}(p) \triangleq \mathit{false}$$

$$\mathcal{D}^{I}_{q}(p) \triangleq D_{c(q)}(p) \vee \exists p'.P_{c(q)}(p, p') \wedge \left(\bigvee_{(q,\psi,q') \in A} \psi(p') \wedge \left(q' \in I \vee \mathcal{D}^{I \cup \{q\}}_{q'}(p') \right) \right)$$

for all $q \in Q$ such that $q \neq q_f$, and where $P_{c(q)}$ and $D_{c(q)}$ are the predicates constructed from the ruleset associated with the node q of the control diagram.

Intuitively, in the final node q_f we accept p as it is, and of course we drop nothing. In all the other nodes, p is accepted as $\tilde{p}$ if and only if there is a path starting from p in the control diagram that obtains $\tilde{p}$ through intermediate transformations. More precisely, for accepting we look for an intermediate packet p', provided that (i) p is accepted as p' by the ruleset $c(q)$ of node q; (ii) p' satisfies one of the predicates ψ labeling the branches of the control diagram; and (iii) p' is accepted as $\tilde{p}$ in the reached node q'. Instead, a packet is dropped directly in the node q or in one of its successors, possibly because it has been translated to p'' via NAT. Note that we ignore paths with loops, because firewalls have mechanisms to detect and discard a packet when its elaboration loops. To this aim, our predicate uses the set I for recording the nodes already traversed.

We conclude this section by establishing the correspondence between the logical formulation and the operational semantics of a firewall. Formally, $\mathcal{F}$ accepts the packet p as $\tilde{p}$ if the predicate $\mathcal{P}_{\mathcal{F}}(p, \tilde{p})$ is satisfied, and vice versa:

Theorem 2.2 (Correctness of the logical characterization)**.** Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and its corresponding predicate $\mathcal{P}_{\mathcal{F}}$, for all packets p we have that $\mathcal{P}_{\mathcal{F}}(p, p') \vee \mathcal{D}_{\mathcal{F}}(p)$ holds and

1. $s \xrightarrow{p,p'} s \uplus (p, p') \implies \mathcal{P}_{\mathcal{F}}(p, p')$

2. $\nexists p', s'.\ s \xrightarrow{p,p'} s' \implies \mathcal{D}_{\mathcal{F}}(p)$

3. $\mathcal{P}_{\mathcal{F}}(p, p') \implies \exists s.s \xrightarrow{p,p'} s \uplus (p, p')$

Recall that the logical characterization abstracts away the notion of state, and thus $\mathcal{P}_{\mathcal{F}}(p, p')$ holds if and only if there exists a state s in which p is accepted as p'. In particular, if the predicate holds for a packet p that belongs to an established connection, p will be accepted only if the relevant state is reached at runtime. This is the usual interpretation of firewall rules for established connections.

Non-deterministic behaviour A configuration can exhibit a non-deterministic behavior: there might exist paths that accept a packet and others that drop it. Such a misbehavior can be detected using the logical characterization. Below, we present a simple example of that.

Example 2.7. Consider the packets p and $\tilde{p}$ and the ruleset R of Example 2.6; let R' consist of $(da(p) = 2 : 22, \texttt{ACCEPT}); (da(p) = 3 : 22, \texttt{DROP})$; and assume that the firewall $\mathcal{F}$ inspects R before R'. The ruleset R accepts p either as $\tilde{p}$ or as $\tilde{p}'$ with $da(\tilde{p}') = 3 : 22$. Consequently, we have that both $\mathcal{P}_{\mathcal{F}}(p, \tilde{p})$ and $\mathcal{D}_{\mathcal{F}}(p)$ are true, since the first rule of R' accepts $\tilde{p}$ and its second one drops $\tilde{p}'$.

Synthesis algorithm

Here, we present an algorithm that synthesizes the fw-function of a firewall configuration $\mathcal{F}$ written in IFCL. This algorithm takes as input the predicates $\mathcal{P}_{\mathcal{F}}$ and $\mathcal{D}_{\mathcal{F}}$ and computes all their models. To do that, we adopt the algorithm of [78], which is based on the Z3 solver [90], and therefore we inherit soundness and completeness. In particular we use their *multi-cubes* representation to succinctly enumerate all the packets accepted and dropped by the firewall. These multi-cubes P are used to succinctly represent the $\tau - pairs$ (P, t) of the fw-function of the firewall where $t \neq \bot$.

We model packets as tuples of Z3 bit-vector variables of appropriate size

```
(srcIP, srcPort, dstIP, dstPort, srcMac, dstMac, protocol, state)
```

that represent source and destination IPs and ports, source and destination MAC addresses, the protocol and the packet state. Firewall predicates are expressed as logical formulas on those packet variables. For example,

$$\texttt{dstIp} \equiv 10.0.2.15 \wedge \texttt{dstPort} == 22$$

selects packets with destination 10.0.2.15 and port 22. We write $\texttt{dstIp} \equiv 10.0.2.15$ as a shortcut for equating `dstIp` with the numerical representation of the IP address 10.0.2.15. Intervals are encoded with two $\leq$ constraints.

Algorithm 1 All-BVSAT*

Require: Formula φ over bit-vectors with free variables $\vec{x}$
Ensure: Set of multi-cubes $\mathcal{M}$ that are models of φ

$B \leftarrow \varphi$
$\mathcal{M} \leftarrow \emptyset$
while B is satisfiable **do**
    $\vec{v} \leftarrow$ a satisfiable assignment to B
    for each multi-cube $\vec{M} \in \mathcal{M}$ **do**
        Extend $\vec{M}$ with $\vec{v}$ if possible
        $B \leftarrow B \wedge (\vec{x} \notin \vec{M})$
    if $B \wedge \vec{x} = \vec{v}$ is still satisfiable **then**
        $\vec{C} \leftarrow \{v_1\} \times ... \times \{v_n\}$
        for each i in $1..n$ **do**
            Expand interval C_i
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{\vec{C}\}$
        $B \leftarrow B \wedge (\vec{x} \notin \vec{M})$
return $\mathcal{M}$

In order to succinctly enumerate packets, a multi-cube maps each packet variable v to a union of disjoint intervals I_v to which the value of v belongs. For instance, the solutions of the formula

$$(\texttt{dstIp} \equiv 10.0.2.15 \ \vee \ \texttt{dstIp} \equiv 10.0.1.0/24) \ \wedge \ (\texttt{dstPort} == 22 \vee \texttt{dstPort} == 443)$$

are expressed by the following multi-cube:

$$\texttt{dstIp} = \{10.0.2.15\} \cup [10.0.1.0, \ 10.0.1.255], \texttt{dstPort} = \{22\} \cup \{443\}$$

For each formula over bit-vector variables, we compute the satisfying multi-cubes using Algorithm 1 of [78]. Intuitively, each iteration of the while loop selects an assignment of variables $\vec{v}$ that is not covered by any of the existing multi-cubes. First the algorithm tries to extend the existing multi-cubes with the values in $\vec{v}$; next, if the formula is still satisfiable, a new multi-cube is created.

During the extension/creation of multi-cubes, the algorithm performs an expansion step that extends as much as possible the intervals both downwards and upwards. This step uses a variant of the binary search algorithm to find the bounds of the maximal interval that satisfies the given formula.

Dealing with NAT Synthesis gets complicated with NAT, because it can introduce many variables in the formulas, representing intermediate address values for the packet during different processing phases. Some variables, however, are not touched by NAT and this needs to be represented in the predicates, as discussed in the following.

An intuitive solution is to impose equality constraints on variables that are not touched by NAT, but this approach may be inefficient. For instance, consider

the formula $1 \leq v_1 \leq 5 \wedge v_1 = v_2$: Algorithm 1 uses the SMT solver to find a solution, e.g., $v_1 = v_2 = \{3\}$, and tries to expand the intervals associated to v_1 and v_2, one after the other. However, increasing the interval of v_1 violates the equality constraint with v_2. The results of the algorithm are thus 5 distinct multi-cubes, i.e., $v_1 = v_2 = \{i \in [1..5]\}$.

A careful treatment of equality introduces new variables only for the packet features that are modified by NAT rules and implicitly model equality constraints by sharing the same variable in the input and in the output packet. For instance, if a NAT rule modifies the destination address of the input packet p, the output packet $\tilde{p}$ is represented with the same variables of p with the exception of the destination address that uses a fresh variable. Since the introduction of these fresh variables is only required for the packets that are subject to NAT, we consider separate predicates covering the different cases: `DNAT`, `SNAT` and filtering. In `DNAT` and `SNAT` all variables will be the same except for the destination and source address, respectively. In filtering, all variables will coincide, as the input and the output packets are the same.

In principle, this separation could lead to an explosion of the number of predicates. However, when studying existing firewall systems, we found that the maximum number of packets to be considered is three: the input packet, the packet after applying destination NAT and the packet after applying source NAT. In fact, in real systems NAT is applied at most twice during packet processing. For this reason, the proposed approach works very well in practice.

2.5 Compilation

The compilation from fw-function to configuration is done in two stages:

1. generation of the IFCL rulesets from the table;
2. association of the IFCL rulesets to the nodes of control diagram of the target language;
3. translation from IFCL to the target language.

However, there are cases in which the compilation fails because the configuration is not expressible in the target system.

The resulting firewall automatically accepts all the packets that belong to established connections with the appropriate translations. This is not a limitation, since it is the default behavior of some real firewall systems (e.g., `pf`) and it is quite odd to drop packets, once the initial connection has been established. Moreover, this is consistent with the over-approximation on the firewall state done in subsection 2.4.1.

We first introduce an algorithm that computes the rulesets of the target firewall $\mathcal{F}_C$ (stage 1). Then, we associate these rulesets with the nodes of its control diagram (stage 2). Finally we discuss the translation from IFCL to the target language (stage 3) and prove the correctness of the compilation.

Algorithm 2 Generation of the rulesets of the compiled firewall $\mathcal{F}_C$

Require: The fw-function τ
Ensure: The rulesets R_{dnat}, R_{fil}, R_{snat}, R_{mark} of the compiled firewall $\mathcal{F}_C$
1: $R_{dnat} = R_{fil} = R_{snat} = R_{mark} = \epsilon$
2: **for** (P, t) in τ **do**
3: **if** $t = id$ **then**
4: add $(\phi_P, \texttt{ACCEPT})$ to R_{fil}
5: **else if** $t \neq \lambda_\perp$ **then**
6: generate fresh tag m
7: add $(\phi_P \wedge tag(p) = \bullet, \texttt{MARK}(m))$ to R_{mark}
8: add $(tag(p) = m, \texttt{NAT}(n_d(t), \star))$ to R_{dnat}
9: add $(tag(p) = m, \texttt{NAT}(\star, n_s(t)))$ to R_{snat}
10: add $(tag(p) \neq \bullet, \texttt{ACCEPT})$ and $(true, \texttt{DROP})$ to R_{fil}
11: prepend R_{mark} to R_{dnat}, R_{fil} and R_{snat}

2.5.1 Ruleset Generation

Algorithm 2 inputs the fw-function τ and generates the basic rulesets R_{fil}, containing filtering, and R_{dnat}, R_{snat} (with default ACCEPT policy) for DNAT and SNAT rules. This separation reflects what is done in all the real systems we have analyzed. Indeed, they can place NAT rules only in specific nodes of their control diagrams, as represented by the labeling in Figure 2.2 e.g., in `iptables`, DNAT is allowed only in rulesets q_1 and q_8, while SNAT only in q_5 and $q_1 1$.

For every $\tau-$pair (P, id), Algorithm 2 generates an accepting rule with predicate ϕ_P, that is verified if the packet is in P (line 4). Also, it produces rules that assign different tags to packets that must be processed by different NAT rules (lines 6 and 7). Each NAT $\tau-$pair is split in a DNAT (line 8) and an SNAT (line 9), where the predicate ϕ is a check on the tag of the packet. We abused notation and used n_d and n_s on a transformation t to select the destination and source part only. Packets subject to NAT are accepted in R_{fil} while the others are dropped (line 10). We prepend R_{mark} to all rulesets making sure that packets are always marked, independently of which ruleset will be processed first (line 11). Recall that the empty tag $\bullet$ identifies untagged packets.

Recall that the @ operator combines rulesets in sequence. Note that R_{fil} drops by default and shadows any ruleset appended to it. In practice, the only interesting rulesets are $R_\epsilon, R_{fil}, R_{dnat}, R_{snat}, R_{dnat} \,@\, R_{fil}, R_{snat} \,@\, R_{fil}$ where R_ϵ is the empty ruleset with default ACCEPT policy.

2.5.2 Ruleset Association

The cap-label associated to a node of the control diagram uniquely determines the ruleset we associate to it. As previously stated, we consider only nodes with cap-label ID, and not associated with both DNAT and SNAT. The correspondence between $V(q)$ and $c(q)$ is thus the following one:

$$V(q) = \{\texttt{ID}\} \Rightarrow c(q) = R_\epsilon \qquad\qquad V(q) = \{\texttt{ID}, \texttt{SNAT}\} \Rightarrow c(q) = R_{snat}$$

$V(q) = \{\texttt{DROP}\} \Rightarrow c(q) = R_{fil}$ $\quad$ $V(q) = \{\texttt{ID}, \texttt{DNAT}, \texttt{DROP}\} \Rightarrow c(q) = R_{dnat} \mathbin{@} R_{fil}$

$V(q) = \{\texttt{ID}, \texttt{DNAT}\} \Rightarrow c(q) = R_{dnat}$ $\quad$ $V(q) = \{\texttt{ID}, \texttt{SNAT}, \texttt{DROP}\} \Rightarrow c(q) = R_{snat} \mathbin{@} R_{fil}$

Example 2.8. Now we map the rulesets to the nodes of the control diagrams of the real systems presented in subsection 2.2.2.
For `iptables` we have:

$$c(q_1) = c(q_8) = R_{dnat} \qquad c(q_5) = c(q_{11}) = R_{snat} \qquad c(q_3) = c(q_6) = c(q_9) = R_{fil}$$

while the remaining nodes get the empty ruleset R_ϵ.
For `pf` we have:

$$c(q_2) = R_{dnat} \qquad c(q_0) = R_{snat} \qquad c(q_1) = R_{fil} \qquad c(q_3) = R_{fil}$$

while the remaining nodes get the empty ruleset R_ϵ.
For `ipfw` we have:

$$c(q_0) = R_{dnat} @ R_{fil} \qquad c(q_1) = R_{snat} @ R_{fil}$$

while the remaining nodes get the empty ruleset R_ϵ.

We now introduce the notion of *compiled firewall*.

Definition 2.11 (Compiled firewall)**.** Let $\mathcal{R}$ be $\{R_\epsilon, R_{fil}, R_{dnat}, R_{snat}, R_{dnat} \mathbin{@} R_{fil}, R_{snat} \mathbin{@} R_{fil}\}$. A firewall $\mathcal{F}_C = (\mathcal{C}, \rho, c)$ with control diagram $\mathcal{C} = (Q, A, q_i, q_f)$ is a *compiled firewall* if

- $c(q_i) = c(q_f) = R_\epsilon$
- $c(q) \in \mathcal{R}$ for all $q \in Q$
- every path π from q_i to q_f in the control diagram $\mathcal{C}$ traverses a node $q \in Q \setminus \{q_i, q_f\}$ such that $c(q) \in \{R_{fil}, R_{dnat} \mathbin{@} R_{fil}, R_{snat} \mathbin{@} R_{fil}\}$

Intuitively, the above definition requires that only the rulesets in $\mathcal{R}$ are associated with the nodes in the control diagram and that all paths pass at least one through a node with the filtering ruleset. Note that the firewalls obtained through the association we have chosen for `iptables`, `pf` and `ipfw` satisfy these conditions.

From IFCL to the target language

Producing the target configuration requires to translate the IFCL firewall obtained by Algorithm 2 into the target language. We proceed like in the encoding phase, presented in subsection 2.2.2, but in the opposite direction. The translation is done in two steps: first the rulesets are separately translated, then they are composed according to the control diagram, hence obtaining the final configuration. Composing the translations of the rulesets is a trivial syntactical work. For the most, this is also the case with the translation of rulesets, but there are some subtleties to consider.

It is worth noting that Algorithm 2 generates a very specific kind of rulesets, characterized by no control flow instructions at all, a heavy usage of tags, and a restricted usage of CHECK-STATE, for accepting all the packets that belong to established connections. Therefore, we do not need a general translation, one that works for this cases suffices.

Rules are usually considered one at a time. The only problem that arises from rule conditions is that concrete languages cannot express arbitrary unions of ranges of addresses. For this reason, the translation has to break the rules having such conditions into a sequence of rules expressing the same intervals, but using subnet notation. In practice, our implementation employs refactoring to reduce the size of the obtained configuration.

Among targets, the only one that causes problems is MARK. This because of two reasons: (i) in some languages (like `iptables`) it is not possible to write an instruction for marking a packet without immediately applying also NAT, ACCEPT, or DROP[1], (ii) some languages (like `pf`), do not allow you to check for the packet to have no tag associated (as in "$tag(p) = \bullet$" of Algorithm 2, line 7).

These problems are solved by postprocessing the compiled firewall rulesets as follows. For (i), before any rule r having condition $tag(p) = \bullet$, we insert a sequence of rules $(tag(p) = m, Goto(R_r))$, one for each m generated by Algorithm 2, where R_r is the continuation of the ruleset after rule r. In this way, the original condition on tag becomes redundant and can be removed. Actually, it is possible to take advantage from the fact that in R_{mark} all the rules containing $tag(p) = \bullet$ are consecutive, and that conditions in every pair of different rules are mutually exclusive, since they came from a fw-function. Thus you can insert the list of $(tag(p) = m, Goto(R_r))$ rules just once, before the first occurrence of $tag(p) = \bullet$.

For addressing (ii), we first move the accepting (in R_{fil}) or translating (in R_{dnat} and R_{snat}) rules immediately after the corresponding tagging rule. Finally, the whole pair (tagging rule, accepting/translating rule) is translated into a single rule of the target language.

Translating the targets NAT, ACCEPT, DROP, and the used instance of CHECK-STATE causes no problems. As we say above, we can leave out CALL(), GOTO() and RETURN, since compiled firewalls do not contain such targets in general form. The simple instances of GOTO() introduced in (i) can be translated trivially in all the considered languages.

Some other postprocessing is introduced for optimization purpose. For example, if there is a set of nodes that are traversed by all the packets only once and before any translation or dropping, as in nodes q_0 and q_7 of `iptables`, then we anticipate the tagging instructions of R_{mark} in these nodes and remove them from the following ones.

[1]This because mark instructions are often implemented as options on rules instead of actual targets on their own.

Correctness of the compiled firewall

We start by showing that a compiled firewall $\mathcal{F}_C$ accepts the *same* packets as the original fw-function τ, possibly with a different translation. The differences may show up because the source and the target firewall systems may impose different constraints on which kinds of packets can be translated, and when.

Theorem 2.3. Let $\mathcal{F}_C$ be a *compiled firewall* of τ and let p be a packet, then

$$\tau(p) \neq \lambda_\perp \Leftrightarrow \exists p'.\, \mathcal{P}_{\mathcal{F}_C}(p, p').$$

It is convenient introducing a few auxiliary definitions. Let $\mathcal{T} = \{id, dnat, snat, nat\}$ be the set of translations of a packet while it traverses a firewall. The first, *id*, represents the identity, *dnat* and *snat* are for DNAT and SNAT, while *nat* represents both. Also, let $(\mathcal{T}, <)$ be the partial order such that $id < dnat$, $id < snat$, $dnat < nat$ and $snat < nat$. Finally, given a packet p and a firewall $\mathcal{F}$, we assume $\pi_{\mathcal{F}}(p)$ to be the unique path in the control diagram of $\mathcal{F}$ along which p is processed. This is the case when NAT to multiple addresses is not allowed, which is the most common situation in practice. The following function computes the *translation capability* of a path π, i.e., which translations can be performed on packets processed along π.

Definition 2.12 (Translation capability)**.** Let $\pi = \langle q_1, \ldots, q_n \rangle$ be a path on the control diagram of a compiled firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$. The *translation capability* of π is

$$tc(\pi) = \text{lub} \bigcup_{q_j \in \pi} \gamma(c(q_j))$$

where lub is the least upper bound operator on $(\mathcal{T}, <)$ and γ is defined as

$$\begin{aligned} \gamma(R) &= \{id\} \text{ for } R \in \{R_\epsilon, R_{fil}\} \\ \gamma(R_t) &= \{t\} \text{ for } t \in \{dnat, snat\} \\ \gamma(R_1 \mathbin{@} R_2) &= \gamma(R_1) \cup \gamma(R_2) \end{aligned}$$

Let $p \approx p'$ hold if and only if $p' = p[tag \mapsto m]$ for some tag m; given a packet p and its translation p', let t_β be defined as follows, where $\beta \in \mathcal{T}$:

$$\begin{aligned} t_{id}(p, p') &= p & t_{dnat}(p, p') &= p[da \mapsto da(p')] \\ t_{nat}(p, p') &= p' & t_{snat}(p, p') &= p[sa \mapsto sa(p')] \end{aligned}$$

The following theorem describes the relationship between a compiled firewall $\mathcal{F}_C$ and the firewall $\mathcal{F}_S$. Intuitively, if $\mathcal{F}_S$ and $\mathcal{F}_C$ both accept a packet p, then the translations applied by $\mathcal{F}_C$ are the ones of $\mathcal{F}_S$ that are also available on the path $\pi_{\mathcal{F}_C}(p)$ in the control diagram of $\mathcal{F}_C$, along which p is processed. This means that the translations of $\mathcal{F}_C$ are as close as possible to the ones of $\mathcal{F}_S$.

Theorem 2.4. Let p be a packet accepted by both $\mathcal{F}_S$ and $\mathcal{F}_C$; let $\beta = tc(\pi_{\mathcal{F}_C}(p))$; and let $p'' \approx t_\beta(p, p')$ for some p'. We have that

$$\mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \mathcal{P}_{\mathcal{F}_C}(p, p'')$$

with $p' = p''$ when $\beta = nat$ or $p = p'$.

Putting together Theorem 2.3 and 2.4, we have that the compiled firewall drops all and only the packet that the source drops, and that all the packet accepted by the source firewall are accepted by the compiled one with a translation that is as close as possible to the original one.

Example 2.9. Consider again Example 2.8. Any path π in `iptables` has $tc(\pi) = nat$, which implies $p' \approx p''$, i.e., $\mathcal{F}_C$ behaves exactly as $\mathcal{F}_S$. Interestingly, the paths $\pi_1 = \langle q_i, q_0, q_1, q_o \rangle$ and $\pi_2 = \langle q_i, q_2, q_3, q_o \rangle$ in `pf` have $tc(\pi)$ equal to *snat* and *dnat*, respectively. In fact, `pf` cannot perform *snat* and *dnat* on packets directed to and generated from the host, respectively. The same holds for `ipfw`.

2.6 The Problem of Expressivity

Here, we introduce two formal notions of expressivity that characterize the set of fw-functions expressible by each language. We provide a logical definition of these notions that will be turned in an operational style and then checked by our implementation. The first notion is *individual expressivity* in which we assume that the firewall manages single packets in isolation. The second is *function expressivity* that characterizes what is expressible, taking care of how a transformation on a packet may affect that of another packet. We only use below the basic targets ACCEPT, DROP and NAT and avoid ad hoc constructs. Indeed, control flow instructions are implicitly considered in the following, because they are macro-expanded in IFCL (see subsection 2.2.5). Tags are also implicitly considered. Indeed, since they can appear everywhere in the rulesets, and are used to distinguish packets, their application allows to consider each packet as if it were managed in isolation by the firewall, as in *individual expressivity*. Conversely, if one prohibits the usage of tags, the resulting expressivity is the one of *function expressivity*. The choice of focusing only on this specific subset of targets is also driven by the empirical evidence that most of the policies freely available use these targets only [49]. Also, we formally relate the expressivity of the languages considered so far, according to these two notions. For convenience, we first define a new, more abstract semantics of IFCL, which is proven equivalent to the operational one.

2.6.1 Denotational Semantics

We first define the semantics of normalized rulesets.

Definition 2.13. Let R be a normalized ruleset. The *semantics* of R is

$$(\!| R |\!) : \mathbb{P} \to \mathcal{T}_{\mathbb{P}}$$

where

$$(\!|\,\epsilon\,|\!)p = \begin{cases} id & \text{if } dp = \texttt{ACCEPT} \\ \lambda_\perp & \text{otherwise} \end{cases}$$

$$(\!|\,(\phi, \texttt{ACCEPT}); R\,|\!)(p) = \begin{cases} id & \text{if } \phi(p) \\ (\!|\,R\,|\!)(p, m) & \text{otherwise} \end{cases}$$

$$(\!|\,(\phi, \texttt{DROP}); R\,|\!)(p) = \begin{cases} \lambda_\perp & \text{if } \phi(p) \\ (\!|\,R\,|\!)(p) & \text{otherwise} \end{cases}$$

$$(\!|\,(\phi, \texttt{NAT}(n_d, n_s)); R\,|\!)(p) = \begin{cases} tr_{nat}(n_d, n_s) & \text{if } \phi(p) \\ (\!|\,R\,|\!)(p) & \text{otherwise} \end{cases}$$

where $tr_{nat}(a_{dIP} : a_{dPort},\ a_{sIP} : a_{sPort}) = (t_{dIP} : t_{dPort},\ t_{sIP} : t_{sPort})$ with

$$t_i = \begin{cases} id & \text{if } a_i = \star \\ \lambda_{a_i} & \text{otherwise} \end{cases}$$

We now define the denotational semantics of the firewalls expressed in normal form. This new semantics is a fw-function, i.e. a function from packets to transformations. The idea is to compose the transformations applied to a packet p in each node of the path it follows in the control diagram.

Definition 2.14. Let $\mathbb{P} \to \mathcal{T}_\mathbb{P}$ be the set of *fw-functions*, and let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall. The *semantics* of $\mathcal{F}$: $\mathbb{P} \to \mathcal{T}_\mathbb{P}$ is defined as

$$(\!|\,\mathcal{F}\,|\!) = \odot^{\mathcal{F}}_{\{q_i\}}(q_i)$$

where for any $I \subseteq Q$

$$\odot^{(C,c)}_I(q)(p) = \begin{cases} \odot^{(C,c)}_{I \cup \{q'\}}(q')(p') \circ t & \text{if } q \neq q_f \wedge t \neq \lambda_\perp \wedge q' \notin I \\ t & \text{if } q = q_f \vee t = \lambda_\perp \\ \lambda_\circlearrowleft & \text{if } q' \in I \end{cases}$$

with $t = (\!|\,c\,|\!)(q)(p)$, $p' = t(p)$ and $q' = \delta(q, p')$.

Note that in the definition above, while traversing the nodes of the control diagram, we not only apply the transformations to packets, but we also accumulate and compose them. Indeed, our goal is to characterize the overall transformation a packet undergoes, rather than simply determine whether it is accepted or not. Recall that in each step the packet p can change and that $\lambda_\perp$ immediately drops it. The special transformation $\lambda_\circlearrowleft$ is applied as soon as a loop is detected, and for that the index I accumulates the nodes already visited. According to our experiments with the real firewall systems described in Section 3.1, cycling packets are usually dropped, i.e. $\lambda_\circlearrowleft = \lambda_\perp$, but no official documentations state this in clear. Thus, best practice suggests to avoid considering configurations where packets cycle. For this reason in the following we only consider firewalls $\mathcal{F}$ where no packets cycle: $(\!|\,\mathcal{F}\,|\!)(p) \neq \lambda_\circlearrowleft$ for any packet p.

2.6.2 Allowed Transformations

In subsection 2.2.3 we presented cap-labels as a way to encode the actions that may appear associated with nodes of a control diagram. Such labels also restrict the *transformations* that may be associated with packets by the semantics of a ruleset in a given node.

Definition 2.15 (Allowed transformations)**.** The mapping ε associates a cap-label $l \in \mathbb{L}$ with the set of allowed transformations as follows:

$$\varepsilon(\texttt{ID}) = \{(id : id\,,\ id : id)\} \qquad \varepsilon(\texttt{DROP}) = \{\lambda_\perp\}$$
$$\varepsilon(\texttt{DNAT}) = \Lambda \times \{id : id\} \qquad \varepsilon(\texttt{SNAT}) = \{id : id\} \times \Lambda$$

where $\Lambda = \{\lambda_a : \lambda_{a'} \mid a \in IP, a' \in Port\}$.

We extend ε to homomorphically operate on sets $L \subseteq \mathbb{L}$ of labels, i.e. returning the union of $\varepsilon(l)$, for $l \in L$.

The following lemma follows immediately by the definitions, and state that a firewall $\mathcal{F}$ is legal if and only if the semantics of the rulesets associated with any node q has a subset of $V(q)$ as image.

Lemma 2.2. A firewall $(\mathcal{C}, \rho, c)$ with nodes Q is *legal for a cap-label assignment* V if and only if for each node $q \in Q$, $\varepsilon(V(q))$ includes the image of $(\!|\, c(q) \,|\!)$.

Consequently, the expressivity of a firewall language $\mathcal{L}$ can be established by just examining the control diagram $\mathcal{C}_\mathcal{L}$ and how the cap-label assignment $V_\mathcal{L}$ constraints the semantics of the rulesets associated with nodes.

For convenience, in the following we will consider *functional firewalls*, i.e. pairs composed by a control diagram $\mathcal{C}$ with nodes Q, and a *functional configuration*, i.e. function $f : : Q \to \mathbb{P} \to \mathcal{T}_\mathbb{P}$.

Definition 2.16 (Functional firewalls)**.** A functional firewall is a pair $(\mathcal{C}, f)$ where $\mathcal{C}$ is a control diagram, and $f : : Q \to \mathbb{P} \to \mathcal{T}_\mathbb{P}$. A functional firewall is *legal for a cap-label assignment* V if and only if for each node $q \in Q$, $\varepsilon(V(q))$ includes the image of $f(q)$.

A trivial consequence of Lemma 2.2 is that a functional firewall $(\mathcal{C}, f)$ is legal for V if and only if a firewall $(\mathcal{C}, \rho, c)$ exists, that is legal for V and such that, for each node $q \in Q$, $f(q) = (\!|\, c(q) \,|\!)$.

2.6.3 Individual Expressivity

The *individual expressivity* of a firewall language describes which transformations a legal configurations can apply to packets. Below we first define it intensionally. We then characterize it constructively in terms of traces, i.e., labeled paths along the control diagram of the language, so providing the bases of the algorithms of Section 2.6.3.

Expressible Pairs

We start with an intensional characterization of the transformations that can be legally applied to each single packet.

Definition 2.17. The set of *expressible pairs* of a language $\mathcal{L}$ is

$$E_{\mathcal{L}} = \{\, (p,t) \mid \exists f.\, f \text{ is legal for } V_{\mathcal{L}} \wedge (\!|\, (C_{\mathcal{L}}, f) \,|\!)(p) = t \,\}$$

We now give a constructive characterization of this expressivity through the notion of trace, i.e., a pair composed by a "complete" acyclic path that a packet may traverse in a control diagram and by the sequence of cap-label of the traversed nodes. A complete path starts from the initial node and either ends in the final node or in one dropping the packet; in addition, for each node q the cap-label is among the permitted ones $V(q)$. Formally,

Definition 2.18. A *trace* of a language $\mathcal{L}$ is a pair $h = (\pi, v)$ where $\pi = q_0 \cdot q_1 \ldots \cdot q_n$, $v = l_0 \cdot l_1 \ldots \cdot l_n$, with $q_0 = q_i$, are such that $\forall i, j.\ i \neq j \Rightarrow q_i \neq q_j$, and $\forall j.\, (q_j, \psi, q_{j+1}) \in A$ for some ψ, $l_j \in V_{\mathcal{L}}(q_j)$, $\forall j \neq n.\, (q_j \neq q_f \wedge l_j \neq \texttt{DROP})$ and either $q_n = q_f$ or $l_n = \texttt{DROP}$. Finally, let $\mathcal{H}_{\mathcal{L}}$ be the set of the traces of $\mathcal{L}$.

By abuse of notation, given a trace (π, v) we call a configuration f *legal* for v when $\forall p \in \mathbb{P}, q_j \in \pi.\, f(q_j)(p) \in \varepsilon(l_j)$.

Example 2.10. The following are traces of `pf`:

$$(q_i \cdot q_0 \cdot q_1 \cdot q_f, \texttt{ID} \cdot \texttt{ID} \cdot \texttt{ID} \cdot \texttt{ID}), \quad (q_i \cdot q_2 \cdot q_3, \texttt{ID} \cdot \texttt{DNAT} \cdot \texttt{DROP})$$

Instead the following two are *not* traces of `pf`:

$$(q_i \cdot q_0 \cdot q_1 \cdot q_f, \texttt{ID} \cdot \texttt{DNAT} \cdot \texttt{ID} \cdot \texttt{ID}), \quad (q_i \cdot q_0 \cdot q_1, \texttt{ID} \cdot \texttt{SNAT} \cdot \texttt{ID})$$

The first is not because of the second cap-label, $\texttt{DNAT} \notin V_{\texttt{pf}}(q_1)$, the second because it is not complete: neither the path ends with q_f nor the sequence of its labels ends with `DROP`.

We now define the *capability of a trace* $h = (\pi, v)$ that describes how a packet may be transformed by the nodes of π. Intuitively, we consider all the possible compositions of the transformations allowed by the cap-labels. Recall that the arcs of control diagrams are guarded by predicates that must be satisfied when a packet flows through them. These predicates determine which packets and transformations must be considered. Given a trace $h = (\pi, v)$, let ψ_j be the predicates labelling the arcs outgoing q_j in the path $q_i \cdot q_1 \ldots \cdot q_n \in \pi$.

Definition 2.19. The *capability of a trace* $h = (\pi, v)$ is the set $\widetilde{E}_h \subseteq \mathbb{P} \times \mathcal{T}_{\mathbb{P}}$, containing the pairs (p, t) such that

$$\exists\, t_1, \ldots t_n.\ t_n \circ \cdots \circ t_1 = t \wedge \forall j.\ t_j \in \varepsilon(l_j) \wedge j < n \rightarrow \psi_j(p_j)$$

where $\forall j.\ p_j = (t_j \circ \cdots \circ t_1)(p)$.
Finally, the *capability of the traces of a language* $\mathcal{L}$ is

$$\widetilde{E}_{\mathcal{L}} = \bigcup_{h \in \mathcal{H}_{\mathcal{L}}} \widetilde{E}_h$$

Example 2.11. Consider the trace of `pf`

$$h = (\pi, v) = (q_i \cdot q_0 \cdot q_1 \cdot q_f,\ \texttt{ID} \cdot \texttt{SNAT} \cdot \texttt{ID} \cdot \texttt{ID})$$

and, assuming that a packet is fully specified by its destination and source addresses, let

$$p_1 = (8.8.8.8 : 22,\ 192.168.0.1 : 50000)$$
$$t_1 = (id : id,\ \lambda_{151.15.1.5} : \lambda_{50000})$$

Then the pair (p_1, t_1) is in $\widetilde{E}_h$, because the transformations

$$id \cdot (id : id,\ \lambda_{151.15.1.5} : \lambda_{50000}) \cdot id \cdot id$$

verify the conditions of Definition 2.19.
Instead, the pair (p_2, t_2) where

$$p_2 = (9.9.9.9 : 22,\ 192.168.0.1 : 50000)$$
$$t_2 = (\lambda_{8.8.8.8} : \lambda_{22},\ id : id)$$

is not in $\widetilde{E}_h$, because no `DNAT` occurs in $v = \texttt{ID} \cdot \texttt{SNAT} \cdot \texttt{ID} \cdot \texttt{ID}$.
Finally, the pair (p_3, t_3) where

$$p_3 = (192.168.0.1 : 80,\ 23.23.23.23 : 50000)$$
$$t_3 = (\lambda_{192.168.0.8} : \lambda_{80},\ id : id)$$

is not in $\widetilde{E}_h$, because $sa(p_3) \notin \mathcal{S}$.

The following theorem assures that a pair (packet, transformation) is expressible by $\mathcal{L}$ if and only if it is expressible by one of its traces h; in other words, the capability of the traces of a language coincides with the set of its expressible pairs. More precisely, Definition 2.17 and 2.19 are equivalent.

Theorem 2.5. $E_{\mathcal{L}} = \widetilde{E}_{\mathcal{L}}$.

Algorithmically Characterizing Individual Expressivity

The operational characterization of individual expressivity is based on Algorithm 3 and on Theorem 2.6 and 2.7 below.

Algorithm 3 checks whether a packet p can follow a given trace $h = (\pi, v)$ by following the path π and verifying if p satisfies all the predicates ψ_j labelling the arc outgoing q_j. It uses the auxiliary functions *length*, *head* and *tail* on sequences with the usual meaning, and the following *Ext* predicate (recall from Definition 2.4 that ψ_w is the predicate ψ restricted on the field w)

$$Ext(\psi, L) = \bigwedge_{w \notin \bigcup_{l \in L} \gamma(l)} \psi_w \qquad \text{where } \gamma(l) = \begin{cases} \{dIP, dPort\} & \text{if } l = \texttt{DNAT} \\ \{sIP, sPort\} & \text{if } l = \texttt{SNAT} \\ W & \text{if } l = \texttt{DROP} \\ \emptyset & \text{if } l = \texttt{ID} \end{cases}$$

A few comments are in order. The cap-labels encountered along the trace h play an important role, and are accumulated in the auxiliary set CL (line 6). Actually, the function Ext weakens the predicate ψ_j by removing the conditions affected by the elements of CL; the resulting predicate is then applied to p to verify if the arc outgoing q_j can be taken (line 8). As soon as the algorithm traverses a node q with cap-label DNAT (SNAT respectively), all the predicates on the destination (source, respectively) addresses are not checked against p any longer. This is because, after leaving q, the original values of the destination fields (source fields, respectively) of p have been changed, thus the relevant predicates will be evaluated on the updated fields. The arc predicates are also accumulated into $\overline{\psi}$ (line 7) and checked for consistency through a call to a SAT procedure (line 8). Intuitively, in a node labeled by SNAT the packet source can always be associated with a value satisfying the constraints of the next arcs in the trace, unless some predicates are contradictory. Since Algorithm 3 considers each node and label of the trace only once, its complexity is linear in the length of the trace, because in the worst case, it visits the nodes in h only once.

The individual expressibility of a language can be characterized by the following theorem, where the function CHECK_FLOW is computed by Algorithm 3, and the function REV(h) reverses the trace h. Note that the packet p traversing the trace h must became $t(p)$ and keep satisfying the predicates on the arcs also after some of its fields have been transformed. If different from those in $t(p)$, the new values of a field can thus be arbitrary chosen, provided that the predicates are satisfied, because the values of the destination (source, respectively) fields between two DNAT (SNAT) are immaterial. Therefore, it suffices checking the destination fields (source, respectively) from the last DNAT (SNAT) label onward. This is actually obtained by reversing the trace h and applying again Algorithm 3. Below, we use the function $\hat{\varepsilon}$ that extends ε to the second component of traces v as follows:

$$\hat{\varepsilon}(l \cdot v) = \hat{\varepsilon}(v) \circ \varepsilon(l)$$

Note that $\hat{\varepsilon}(v) = \lambda_\perp$ if the last element of v is DROP.

Theorem 2.6. Given a trace $h = (\pi, v)$, the pair (p, t) is in $\tilde{E}_h$ iff

$$t \in \hat{\varepsilon}(v) \wedge \text{CHECK\_FLOW}(h, p) \wedge \text{CHECK\_FLOW}(\text{REV}(h), t(p))$$

Example 2.12. Consider the pair (p_1, t_1) and the trace h of Example 2.11. To make sure that $(p_1, t_1) \in E_{\mathcal{H}}(h)$, we first check that $t_1 \in \hat{\varepsilon}(v) = \varepsilon(\texttt{SNAT})$; then, CHECK_FLOW(h, p_1) returns *true* because $sa(p_1) \in \mathcal{S}$ and $da(p_1) \notin \mathcal{S}$; finally, CHECK_FLOW(REV(h), p_1) is also *true* because the fields changed by t_1 do not appear on the arcs of the path of p_1.

The pair (p_3, t_3) instead is not in $E_{\mathcal{H}}(h)$, because $sa(p_3) \notin \mathcal{S}$.

Comparing the Individual Expressivity of Languages

To compare the individual expressivity of different languages, we finitely enumerate their expressible pairs. For that, we use the intuition behind Algorithm 3

Algorithm 3 Checks if a packet p may follow the trace $h = (\pi, v)$

```
1:  function CHECK_FLOW(p, (π, v))
2:      CL ← ∅
3:      ψ̄ ← true
4:      while length(π) > 1 do
5:          (q_j, l_j) ← (head(π), head(v))
6:          CL ← CL ∪ {l_j}
7:          ψ̄ ← Ext(ψ̄, l_j) ∧ ψ_j
8:          if ¬(Sat(ψ̄) ∧ Ext(ψ_j, CL)(p)) then return false
9:          (π, v) ← (tail(π), tail(v))
10:     return true
```

and Theorem 2.6 to partition the set of pairs (p, t) in equivalence classes ω so that if $(p_1, t_1) \in \omega$ is expressible (not expressible, respectively), then all the pairs in ω are expressible (not expressible, respectively) as well. As we will see, a language can have different such equivalence classes, which can be efficiently determined. Since their number is small, one can easily enumerate the expressible pairs of different languages and compare them. Formally:

Definition 2.20. Let $\mathbb{T} = \{\varepsilon(\texttt{ID}), \varepsilon(\texttt{DNAT}), \varepsilon(\texttt{SNAT}), \varepsilon(\texttt{DROP}), \Lambda \times \Lambda\}$ be a partition of the set of transformations $\mathcal{T}_{\mathbb{P}}$; let $\Psi_{\mathcal{L}}$ be the set of the predicates labeling the arcs of the control diagram of a given language $\mathcal{L}$; and let $g \colon \mathbb{P} \to 2^{\Psi_{\mathcal{L}}}$ be defined as

$$g(p) = \{\psi \in \Psi_{\mathcal{L}} \mid p \text{ satisfies } \psi\}$$

Then, for any X_1 and X_2 subsets of $\Psi_{\mathcal{L}}$ and for any $Y \in \mathbb{T}$, let Ω contain the following sets of pairs (p, t)

$$\omega_{X_1,Y,X_2} = \{(p,t) \mid t \in Y \wedge g(p) = X_1 \wedge \big(t(p) \neq \perp \Rightarrow g(t(p)) = X_2\big)\}$$

Intuitively, two pairs (p, t) and (p', t') belong to the same equivalence class if and only if p and p' satisfy the *same* set of predicates X_1; their (not dropping) transformations $t(p)$ and $t'(p')$ also satisfy the *same* predicates X_2; and both t and t' are represented by the *same* sequence of cap-labels.

Example 2.13. The two pairs (p_1, t_1) and (p_2, t_2) where

$$\begin{aligned} p_1 &= (8.8.8.8 : 80,\ 192.168.0.1 : 50000) \\ t_1 &= (\lambda_{7.7.7.7} : \lambda_{80},\ id : id) \\ p_2 &= (9.9.9.9 : 80,\ 192.168.0.1 : 50000) \\ t_2 &= (\lambda_{151.15.1.5} : \lambda_{80},\ id : id) \end{aligned}$$

are not in the same equivalence class.
Indeed, even though $g(p_1) = g(p_2) = \{sa(p) \in \mathcal{S}, da(p) \notin \mathcal{S}\}$, and $t_1, t_2 \in \varepsilon(\texttt{DNAT})$, we have that $t_1(p_1)$ verifies $da(p) \notin \mathcal{S}$ whereas $t_2(p_2)$ does not, hence $g(t_1(p_1)) \neq g(t_2(p_2))$.

We have the following theorem:

Theorem 2.7. Ω is a partition of $\mathbb{P} \times \mathcal{T}_{\mathbb{P}}$ such that the elements of ω_{X_1,Y,X_2} are either all expressible or all not expressible.

Using the equivalence classes we succinctly enumerate and compare the expressible pairs of firewall languages, summarized in Table 2.3. The first three columns show the equivalence class under consideration; the last columns describe whether the pair representative for each class is expressible or not by a language. The details for selecting a representative pair of a given class are in Section A.4.

Note that the predicates on the arcs of the control diagrams of `iptables`, `ipfw` and `pf` only check the *IP* addresses (see subsection 2.2.2). A packet can traverse an arc depending on whether the source or destination of the packet belongs to $\mathcal{S}$. Therefore, we simply use the set $\Psi_\star = \{da(p) \in \mathcal{S}, da(p) \notin \mathcal{S}, sa(p) \in \mathcal{S}, sa(p) \notin \mathcal{S}\}$, the satisfiable subsets of which follow:

$$\{da(p) \in \mathcal{S}, sa(p) \in \mathcal{S}\} \qquad \{da(p) \in \mathcal{S}, sa(p) \notin \mathcal{S}\}$$
$$\{da(p) \notin \mathcal{S}, sa(p) \in \mathcal{S}\} \qquad \{da(p) \notin \mathcal{S}, sa(p) \notin \mathcal{S}\}$$

The theorem below states that only IFCL and `iptables` express all the pairs, while `pf` and `ipfw` have the same expressive power. Its proof directly follows by inspecting Table 2.3.

Theorem 2.8. $E_{\texttt{pf}} = E_{\texttt{ipfw}} \subsetneq E_{\texttt{iptables}} = E_{IFCL} = \mathbb{P} \times \mathcal{T}_{\mathbb{P}}$

2.6.4 Function Expressivity

Individual expressivity only considers packets in isolation. However, a firewall handles many different packets at the same time, each subject to different transformations. The interaction of the transformations applied to different packets must thus be considered. Surprisingly, this affects the ability of the various languages to define configurations, as shown by Example 2.14 below. We first say that a fw-function τ is expressible by a language if the semantics of one of its firewalls is τ.

Definition 2.21. Given a language $\mathcal{L}$ with control diagram $C_{\mathcal{L}}$ and a cap-label assignment $V_{\mathcal{L}}$, a fw-function $\tau\colon \mathbb{P} \to \mathcal{T}_{\mathbb{P}}$ is *$\mathcal{L}$-expressible* iff $\exists f$ legal for $V_{\mathcal{L}}$ such that $(\!|\, (C_{\mathcal{L}}, f)\,|\!) = \tau$.
Call $T_{\mathcal{L}}$ the set of the fw-functions expressible by $\mathcal{L}$.

Note that this notion differs from individual expressivity because it considers *all* the pairs $(p, \tau(p))$ at the same time. The following theorem gives a necessary condition for function expressibility.

Theorem 2.9. Given a language $\mathcal{L}$ and a fw-function τ

$$\tau \in T_{\mathcal{L}} \text{ only if } \forall p \in \mathbb{P}.\, (p, \tau(p)) \in E_{\mathcal{L}}.$$

Table 2.3: The expressible pairs of `iptables`, `pf` and `ipfw`. The first two columns contain the predicates in the subset X_1 (the $-$ stands for both $\in$ and $\notin \mathcal{S}$); the third column contains the set of transformations Y; the fourth and fifth columns contain the predicates in the subset X_2; the sixth column contains the representative pair (p, t) for the given class or $\square$ if the class is empty; the other columns have ✓ if the class containing (p, t) is expressible or ✗ if not.

	X_1		Y	X_2		(p,t)	$E_{\mathcal{L}}$	
	$da(p)$	$sa(p)$		$da(t(p))$	$sa(t(p))$	$(\,(da(p), sa(p)),\, t\,)$	`pf/ipfw`	`iptables`
1	$\in \mathcal{S}$	$\in \mathcal{S}$	ε(`ID`)	$\in \mathcal{S}$	$\in \mathcal{S}$	$((a:r, a:r), id)$	✓	✓
2	$\in \mathcal{S}$	$\notin \mathcal{S}$	ε(`ID`)	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, b:r), id)$	✓	✓
3	$\notin \mathcal{S}$	$\in \mathcal{S}$	ε(`ID`)	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((b:r, a:r), id)$	✓	✓
4	$\notin \mathcal{S}$	$\notin \mathcal{S}$	ε(`ID`)	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, b:r), id)$	✓	✓
5	$-$	$\in \mathcal{S}$	ε(`ID`)	$-$	$\notin \mathcal{S}$	$\square$		
6	$\in \mathcal{S}$	$-$	ε(`ID`)	$\notin \mathcal{S}$	$-$	$\square$		
7	$\in \mathcal{S}$	$\in \mathcal{S}$	ε(`DNAT`)	$\in \mathcal{S}$	$\in \mathcal{S}$	$((a:r, a:r), (\lambda_a : \lambda_r, id : id))$	✓	✓
8	$\in \mathcal{S}$	$\in \mathcal{S}$	ε(`DNAT`)	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((a:r, a:r), (\lambda_b : \lambda_r, id : id))$	✗	✓
9	$\in \mathcal{S}$	$\notin \mathcal{S}$	ε(`DNAT`)	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, b:r), (\lambda_a : \lambda_r, id : id))$	✓	✓
10	$\in \mathcal{S}$	$\notin \mathcal{S}$	ε(`DNAT`)	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, b:r), (\lambda_b : \lambda_r, id : id))$	✓	✓
11	$\notin \mathcal{S}$	$\in \mathcal{S}$	ε(`DNAT`)	$\in \mathcal{S}$	$\in \mathcal{S}$	$((b:r, a:r), (\lambda_a : \lambda_r, id : id))$	✗	✓
12	$\notin \mathcal{S}$	$\in \mathcal{S}$	ε(`DNAT`)	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((b:r, a:r), (\lambda_b : \lambda_r, id : id))$	✗	✓
13	$\notin \mathcal{S}$	$\notin \mathcal{S}$	ε(`DNAT`)	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, b:r), (\lambda_a : \lambda_r, id : id))$	✓	✓
14	$\notin \mathcal{S}$	$\notin \mathcal{S}$	ε(`DNAT`)	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, b:r), (\lambda_b : \lambda_r, id : id))$	✓	✓
15	$-$	$\in \mathcal{S}$	ε(`DNAT`)	$-$	$\notin \mathcal{S}$	$\square$		
16	$-$	$\notin \mathcal{S}$	ε(`DNAT`)	$-$	$\in \mathcal{S}$	$\square$		
17	$\in \mathcal{S}$	$\in \mathcal{S}$	ε(`SNAT`)	$\in \mathcal{S}$	$\in \mathcal{S}$	$((a:r, a:r), (id : id, \lambda_a : \lambda_r))$	✓	✓
18	$\in \mathcal{S}$	$\in \mathcal{S}$	ε(`SNAT`)	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, a:r), (id : id, \lambda_b : \lambda_r))$	✓	✓
19	$\in \mathcal{S}$	$\notin \mathcal{S}$	ε(`SNAT`)	$\in \mathcal{S}$	$\in \mathcal{S}$	$((a:r, b:r), (id : id, \lambda_a : \lambda_r))$	✗	✓
20	$\in \mathcal{S}$	$\notin \mathcal{S}$	ε(`SNAT`)	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, b:r), (id : id, \lambda_b : \lambda_r))$	✗	✓
21	$\notin \mathcal{S}$	$\in \mathcal{S}$	ε(`SNAT`)	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((b:r, a:r), (id : id, \lambda_a : \lambda_r))$	✓	✓
22	$\notin \mathcal{S}$	$\in \mathcal{S}$	ε(`SNAT`)	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, a:r), (id : id, \lambda_b : \lambda_r))$	✓	✓
23	$\notin \mathcal{S}$	$\notin \mathcal{S}$	ε(`SNAT`)	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((b:r, b:r), (id : id, \lambda_a : \lambda_r))$	✓	✓
24	$\notin \mathcal{S}$	$\notin \mathcal{S}$	ε(`SNAT`)	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, b:r), (id : id, \lambda_b : \lambda_r))$	✓	✓
25	$\in \mathcal{S}$	$-$	ε(`SNAT`)	$\notin \mathcal{S}$	$-$	$\square$		
26	$\notin \mathcal{S}$	$-$	ε(`SNAT`)	$\in \mathcal{S}$	$-$	$\square$		
27	$\in \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\in \mathcal{S}$	$((a:r, a:r), (\lambda_a : \lambda_r, \lambda_a : \lambda_r))$	✓	✓
28	$\in \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, a:r), (\lambda_a : \lambda_r, \lambda_b : \lambda_r))$	✓	✓
29	$\in \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((a:r, a:r), (\lambda_b : \lambda_r, \lambda_a : \lambda_r))$	✗	✓
30	$\in \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, a:r), (\lambda_b : \lambda_r, \lambda_b : \lambda_r))$	✗	✓
31	$\in \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\in \mathcal{S}$	$((a:r, b:r), (\lambda_a : \lambda_r, \lambda_a : \lambda_r))$	✗	✓
32	$\in \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, b:r), (\lambda_a : \lambda_r, \lambda_b : \lambda_r))$	✗	✓
33	$\in \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((a:r, b:r), (\lambda_b : \lambda_r, \lambda_a : \lambda_r))$	✓	✓
34	$\in \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((a:r, b:r), (\lambda_b : \lambda_r, \lambda_b : \lambda_r))$	✓	✓
35	$\notin \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\in \mathcal{S}$	$((b:r, a:r), (\lambda_a : \lambda_r, \lambda_a : \lambda_r))$	✗	✓
36	$\notin \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, a:r), (\lambda_a : \lambda_r, \lambda_b : \lambda_r))$	✗	✓
37	$\notin \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((b:r, a:r), (\lambda_b : \lambda_r, \lambda_a : \lambda_r))$	✗	✓
38	$\notin \mathcal{S}$	$\in \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, a:r), (\lambda_b : \lambda_r, \lambda_b : \lambda_r))$	✗	✓
39	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\in \mathcal{S}$	$((b:r, b:r), (\lambda_a : \lambda_r, \lambda_a : \lambda_r))$	✗	✓
40	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\in \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, b:r), (\lambda_a : \lambda_r, \lambda_b : \lambda_r))$	✗	✓
41	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\in \mathcal{S}$	$((b:r, b:r), (\lambda_b : \lambda_r, \lambda_a : \lambda_r))$	✓	✓
42	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$\Lambda \times \Lambda$	$\notin \mathcal{S}$	$\notin \mathcal{S}$	$((b:r, b:r), (\lambda_b : \lambda_r, \lambda_b : \lambda_r))$	✓	✓
43	$\in \mathcal{S}$	$\in \mathcal{S}$	ε(`DROP`)	$-$	$-$	$((a:r, a:r), \bot)$	✓	✓
44	$\in \mathcal{S}$	$\notin \mathcal{S}$	ε(`DROP`)	$-$	$-$	$((a:r, b:r), \bot)$	✓	✓
45	$\notin \mathcal{S}$	$\in \mathcal{S}$	ε(`DROP`)	$-$	$-$	$((b:r, a:r), \bot)$	✓	✓
46	$\notin \mathcal{S}$	$\notin \mathcal{S}$	ε(`DROP`)	$-$	$-$	$((b:r, b:r), \bot)$	✓	✓

The following example shows that the condition above is not a sufficient one.

Example 2.14. The fw-function τ defined below is not `pf`-expressible, although all the pairs $(p, \tau(p))$ are expressible in `pf`.

$$\tau(p)=\begin{cases}(\lambda_{8.8.8.8}:id, \lambda_{151.15.1.5}:id) & \text{if } p_{sIP}=192.168.0.8 \wedge \\ & \quad p_{dIP}=6.6.6.6 \\ (\lambda_{8.8.8.8}:id, id:id) & \text{if } p_{sIP}=192.168.0.8 \wedge \\ & \quad p_{dIP}=7.7.7.7 \\ \lambda_{\perp} & \text{otherwise}\end{cases}$$

All the pairs $(p, \tau(p))$ are expressible in `pf` (in subsection 2.6.5 we present an algorithm to check that this is actually the case). To show that function τ is not expressible by `pf`, we assume by contradiction that it is. Take two packets p and p', both with source IP 192.168.0.8, that only differ on their destination IP which are 6.6.6.6 for p and 7.7.7.7 for p'. They must traverse the nodes q_i, q_2, q_3, q_0, q_1 and q_f, and both are transformed by `DNAT` in q_2 into p'' with destination IP 8.8.8.8. When p and p' arrive in node q_0 they have been already transformed in p'', and when applying `SNAT` the two cannot be taken apart. But τ says that one has to be subject to `SNAT` and the other has not.

Function expressivity enables us to further compare `iptables`, `ipfw` and `pf` that originate a partial order, the top of which is IFCL that can express all the fw-functions.

Theorem 2.10.

- $T_{\texttt{pf}} \subsetneq T_{\texttt{ipfw}} \subsetneq T_{IFCL}$
- $T_{\texttt{iptables}} \subsetneq T_{IFCL}$
- $T_{\texttt{pf}} \nsubseteq T_{\texttt{iptables}}$, $T_{\texttt{iptables}} \nsubseteq T_{\texttt{pf}}$
- $T_{\texttt{ipfw}} \nsubseteq T_{\texttt{iptables}}$, $T_{\texttt{iptables}} \nsubseteq T_{\texttt{ipfw}}$

2.6.5 Checking the expressivity of a fw-function

In this section we describe an algorithm for checking if a policy can be expressed by a given firewall language.

Preprocessing of the fw-function

In addition, we make sure that every τ-pair (P_i, t) is such that all (p, t) with $p \in P_i$ belong to the same equivalence class of Ω. This further partition of τ-pairs is based on the fact that each subset $\omega_{X_1,Y,X_2} \in \Omega$ can be represented as $\{(P_{X_1,t,X_2}, t) | t \in Y\}$ lifting Definition 2.20 to sets of packets and taking a single transformation t. We can then componentwise split each (P_i, t) into its intersections with the pairs (P_{X_1,t,X_2}, t).

Algorithm 4 Check function expressivity of $\mathcal{L}$

function CHECK_FUNCTION(τ, C, V)
  for all $q \in Q$ **do** $g(q) \leftarrow \emptyset$
  for all $(P,t) \in \tau$ **do**
    $h \leftarrow$ COMPUTE_TRACE$(C, V, ([P], t))$
    if $h =$ Null **then** *print* (P,t) not expressible
    else $g \leftarrow$ CHECK_PAIR$(h, (P,t), g)$

function COMPUTE_TRACE$(C, V, (p,t))$
  for all $h \in \mathcal{H}_{\mathcal{L}}$ **do**
    if $t \in \hat{\varepsilon}(v) \wedge$ CHECK_FLOW$(h, p) \wedge$
      CHECK_FLOW(REV$(h), t(p))$ **then return** h
  return Null

function CHECK_PAIR(h, (P,t), g)
  $(P_@, t_\triangleright, t_\triangleleft) \leftarrow (P, t, (id:id, id:id))$
  for all $(q,l) \in h$ **do**
    $(t_@, t_\triangleright) \leftarrow$ SPLIT$(t_\triangleright, l)$
    for all $((\tilde{P}_@, \tilde{t}_\triangleleft, \tilde{t}_@), (\tilde{P}, \tilde{t})) \in g(q)$ s.t. $\tilde{t}_@ \neq t_@$ **do**
      $P^\times_@ \leftarrow P_@ \cap \tilde{P}_@$
      if $P^\times_@ \neq \emptyset$ **then**
        $P^\times \leftarrow P \cap t_\triangleleft^{-1}(P^\times_@)$
        $\tilde{P}^\times \leftarrow \tilde{P} \cap \tilde{t}_\triangleleft^{-1}(P^\times_@)$
        print $(P^\times, t), (\tilde{P}^\times, \tilde{t})$ clash in $q : (P^\times_@, t_@, \tilde{t}_@)$
    $g(q) \leftarrow g(q) \cup \{((P_@, t_\triangleleft, t_@), (P,t))\}$
    $t_\triangleleft \leftarrow t_@ \circ t_\triangleleft$
    $P_@ \leftarrow t_@(P_@)$
  return g

After the steps above, given a τ-pair (P,t) we can pick up a single packet $p \in P$, denoted by $[P]$, with the guarantee that any other packet in P is transformed in the same manner and follows the same trace.

Example 2.15. Consider the following fw-function τ, where *Internet* stands for any public address not in the protected LAN.

$$\tau(p)=\begin{cases} (\lambda_{192.168.0.6} : id, id : id) & \text{if } p_{sIP} \in \mathit{Internet} \wedge \\ & \quad p_{dIP} = 151.15.1.5 \wedge p_{dPort} = 22 \\ \lambda_\perp & \text{if } p_{sIP} \in \mathit{Internet} \wedge \\ & \quad (p_{dIP} \in 192.168.0.0/24 \vee \\ & \quad p_{dPort} \neq 22) \\ (id : id, id : id) & \text{otherwise} \end{cases}$$

The set of packets to be dropped is not a multi-cube, and is represented as the

union of $P_{\perp 1} = 192.168.0.0/24 : \_ \times \mathit{Internet} : \_$ and $P_{\perp 2} = \_ : \_ \times \mathit{Internet} : [0-21] \cup [23-65536]$, where $\_$ stands for "any value." Note also that $(P_{\perp 1}, \lambda_\perp)$ is not included in any equivalence class, and thus it is split into ([192.168.0.2, 192.168.0.255] : $\_ \times \mathit{Internet} : \_$, $\lambda_\perp$) and ($\{192.168.0.1\} : \_ \times \mathit{Internet} : \_$, $\lambda_\perp$) (we omit here the invalid address 192.168.0.0).

Mechanically Checking Expressivity

We now describe Algorithm 4. Roughly, it iterates on all the τ-pairs (P, t) representing a fw-function τ. For each pair, it creates an *extended configuration* g that associates each node q with the required transformation t_q (occurring in the cap-labels of q). In doing so, the algorithm checks that no *clash* occurs. Intuitively, a configuration has clashes when in a given node of the control diagram it prescribes to apply "incompatible" transformations to a packet, e.g., drop and transform it at the same time, or change its destination field to two different addresses. Formally,

Definition 2.22 (Clashes)**.** Given a language $\mathcal{L}$, two pairs (p, t) and $(\tilde{p}, \tilde{t})$ collide in a node $q \in C_\mathcal{L}$ with clash $(p_@, t_@, \tilde{t}_@)$ iff, for all configurations f legal for $V_\mathcal{L}$

$$(\!|\, (C_\mathcal{L}, f) \,|\!)(p) = t \;\Rightarrow\; f(q)(p_@) = t_@$$
$$(\!|\, (C_\mathcal{L}, f) \,|\!)(\tilde{p}) = \tilde{t} \;\Rightarrow\; f(q)(p_@) = \tilde{t}_@$$

and $\tilde{t}_@ \neq t_@$.
Also, we say that two τ-pairs (P, t) and $(\tilde{P}, \tilde{t})$ collide in $q \in C_\mathcal{L}$ with clash $(P_@, t_@, \tilde{t}_@)$ iff, for all configurations f legal for $V_\mathcal{L}$

$$(\ \forall p \in P.\ (\!|\, (C_\mathcal{L}, f) \,|\!)(p) = t\ ) \;\Rightarrow\; \forall p_@ \in P_@.\ f(q)(p_@) = t_@$$
$$(\ \forall \tilde{p} \in \tilde{P}.\ (\!|\, (C_\mathcal{L}, f) \,|\!)(\tilde{p}) = \tilde{t}\ ) \;\Rightarrow\; \forall p_@ \in P_@.\ f(q)(p_@) = \tilde{t}_@$$

and $\tilde{t}_@ \neq t_@$.

The CHECK_FUNCTION in Algorithm 4 takes as input an fw-function τ and a firewall language $\mathcal{L}$, represented by its control diagram C and the cap-label assignment V. First, the function initializes the extended configuration g we want to build as "empty" and then iterates the following steps over all τ-pairs (P, t) of τ. For each τ-pair (P, t) we look for a trace h of C that can express it. If there is none, the transformation t cannot be associated with the packets represented by P (line 5). Otherwise, two cases may arise. The first is when two packets processed in a node q of C clash, as detailed below (line 17). The other possible case is when the configuration g can be correctly updated with the new τ-pair (P, t).

The auxiliary function COMPUTE_TRACE returns a trace that can expresses (p, t), if any, when $p = [P]$ is one of the packets of P (cf. Theorem 2.6).

The auxiliary function CHECK_PAIR is called with a trace h returned by COMPUTE_TRACE; a τ-pair; and an (incomplete) extended configuration g. An

extended configuration maps each node to a pair and intuitively extends a configuration with the transformations annotated by the τ-pair in which they occur. Roughly, CHECK_PAIR visits sequentially each node q of h; computes the transformations that are applied in it; and updates the configuration g accordingly. More precisely, CHECK_PAIR computes the transformations $t_\triangleleft$ and $t_@$, and a new multi-cube $P_@$ for each node q. The transformation $t_\triangleleft$ describes how the packets have been rewritten along the sub-path of h from q_i to q. Whereas $t_@$ is the transformation to be applied in q to packets in the multi-cube $P_@$, obtained from the initial one P applying $t_\triangleleft$. For each node, $t_@$ is extracted from the transformation $t_\triangleright$ that records the part of t still to be considered. We get $t_@$ through the SPLIT function, a sort of inverse of function composition, that given a $t_\triangleright$ and a cap-label l returns the transformation satisfying l and removes it from $t_\triangleright$. Summing up, each node q is associated with pairs of the form $((P_@, t_\triangleleft, t_@), (P, t))$. Suppose now that a node is associated with two such pairs, one with $t_@$ to be applied to $P_@$ and the other with a different transformation $\tilde{t}_@$ to be applied to $\tilde{P}_@$. A clash occurs if the two multi-cubes have non-empty intersection (line 16). Indeed, the packets in the intersection will be transformed in two conflicting ways by $t_@$ and $\tilde{t}_@$, and the clash is reported to user (line 17). The lines 18 and 19 recover the packets of P and $\tilde{P}$ clashing in q, by computing the pre-image of $P_@^\times$ under the transformation applied from q_i to q. The last three lines update $g, t_\triangleleft$ and $P_@$, respectively.

Example 2.16. Consider the following two τ-pairs, where *Internet* stands for any public address not in the protected LAN, and _ stands for "any port"

$$
\begin{aligned}
&(P_1, t_1) = \\
&\quad (\{151.15.1.5\} : \{22\} \times \mathit{Internet} : \_,\ (\lambda_{192.168.0.6} : id, id : id)) \\
&(P_2, t_2) = \\
&\quad ([192.168.0.2, 192.168.0.255] : \_ \times \mathit{Internet} : \_,\ \lambda_\perp)
\end{aligned}
$$

Choose $[P_1] = (151.15.1.5 : 22 \times 6.6.6.6 : 55)$ as the representative for P_1. It follows the trace $([q_i, q_2, q_3, q_f], [\texttt{ID}, \texttt{DNAT}, \texttt{ID}, \texttt{ID}])$. After evaluating q_3, the call to CHECK_PAIR results in the extended configuration g where $\overrightarrow{id}$ stands for $(id : id, id : id)$

$$
\begin{aligned}
g(q_i) &= (P_1, \overrightarrow{id}, \overrightarrow{id}), (P_1, t_1) \\
g(q_2) &= (P_1, \overrightarrow{id}, t_1), (P_1, t_1) \\
g(q_3) &= ((\{192.168.0.6\} : \{22\} \times \mathit{Internet} : \_\ ), t_1, \overrightarrow{id}), (P_1, t_1)
\end{aligned}
$$

In the same way, choose $[P_2] = (192.168.0.3 : 11 \times 7.7.7.7 : 44)$ that follows the trace $([q_i, q_2, q_3], [\texttt{ID}, \texttt{ID}, \texttt{DROP}])$. When q_2 is reached, the call to CHECK_PAIR results in updating g as follows

$$
\begin{aligned}
g(q_i) &= g(q_i) \cup (P_2, \overrightarrow{id}, \overrightarrow{id}), (P_2, t_2) \\
g(q_2) &= (P_2, \overrightarrow{id}, \overrightarrow{id}), (P_2, t_2)
\end{aligned}
$$

When $(q_3, \texttt{DROP})$ is considered, $P^{\times}_{@} = (P_{@} \cap \tilde{P}_{@}) = \tilde{P}_{@} \neq \emptyset$, where $P_{@} = P_2$ and $\tilde{P}_{@} = (\{192.168.0.6\} : \{22\} \times \mathit{Internet} : \_\,)$, making (P_1, t_1) and $((\{192.168.0.6\} : \{22\} \times \mathit{Internet} : \_\,), t_2)$ to clash in q_3.

To compute the cost of Algorithm 4 we consider the specific policy τ on which it runs. More prcisely, in the formula below we take care of the number: of $\mathcal{H}_{\mathcal{L}}$, the traces of $\mathcal{L}$; of $Q_{\mathcal{L}}$, the nodes in the control diagram of $\mathcal{L}$; of the τ-pairs of τ; and of the intervals in the field $w \in W$ of P. For each τ-pair (P, t), the algorithm inspects the traces of $\mathcal{L}$ to select h, the one that expresses (P, t). For that, Algorithm 3 is invoked that requires $Q_{\mathcal{L}}$ iterations at most. Then, for each node of h (containing all the nodes of $Q_{\mathcal{L}}$ in the worst case) the intersections in lines 16, 18, and 19 are computed between the actual pair and those that visit the same node (all the other τ-pairs in the worst case). Finally, the intersection between two multicubes P and $\tilde{P}$ requires intersecting the intervals P_w and $\tilde{P}_w$ for each field $w \in W$. Summing up, we obtain the following formula:

$$|\tau| \cdot |Q_{\mathcal{L}}| \cdot (|\mathcal{H}_{\mathcal{L}}| + |\tau| \cdot |W| \cdot max\{|P_w| \mid (P, t) \in \tau\}).$$

Algorithm 4 is correct, as stated below. However, its correctness relies on two assumptions that are verified by all the languages that we have studied so far, including `iptables`, `pf` and `ipfw`. The first is that each pair (p, t) can be expressed by no more than one trace $h \in \mathcal{H}_{\mathcal{L}}$. The only exception is when $t = \lambda_{\perp}$. We choose to only keep the traces $(q_i \ldots q_n), (l_i \ldots \texttt{DROP})$ such that $\forall j < n. \texttt{DROP} \notin V(q_j)$ and $l_j = \texttt{ID}$ (recall that `ID` is associated with every node of C). Roughly, these traces are the shortest that drop a packet without transforming any of its fields. Taking the shortest traces does not affect the expressivity of a language, and there is no point in changing discarded packets. The other assumption is that the trace h contains no repeated `DNAT` or `SNAT` labels, and makes the extraction of $t_{@}$ from $t_{\triangleright}$ well-defined, in particular when $t = \lambda_{\perp}$.

Theorem 2.11. For each firewall language $\mathcal{L}$ and fw-function τ, the Algorithm 4 is correct because it prints all and only

1. the τ-pairs (P, t) not expressible by $\mathcal{L}$;
2. the τ-pairs (P, t) and $(\tilde{P}, \tilde{t})$ that clash on some node q.

This theorem has some important practical consequences, as granted by the following corollary.

Corollary 2.1. A fw-function τ is expressible by $\mathcal{L}$ if and only if Algorithm 4 prints nothing.

Thus, if the administrators solve all the inexpressible pairs and the reported clashes, then they obtain a configuration for the desired system. The inexpressible pairs can be simply removed if they are not relevant, otherwise the administrator can patch the configuration through calls to external code, e.g.,

NFQUEUE target in `iptables` [10]. There are two different ways to solve clashes, depending on whether the intended behaviour of the system is implemented. One is selecting the more appropriate transformation $t_{@}$ and $\tilde{t}_{@}$ for every clashing t-pairs. Actually, acting on the transformations may change the semantics of the firewall. The other solution is semantics-preserving: one may use other features of the language to distinguish between the two clashing sets of packets, e.g., using tags or external code. In Section 2.7.2 we show on an example that our implementation alerts an administrators when this is the case, and proposes a tag-based solution.

2.7 Implementation

Our approach for managing firewalls is validated by two tools: FWS that implements the compilation and decompilation, and *F2F* that checks the expressivity of a fw-function in a given firewall language. Below, we first describe them, then we validate them on real-world configurations.

2.7.1 FWS

The tool FWS [13] allows the administrator to (*i*) decompile a configuration obtaining a fw-function displayed as a table, and (*iii*) compile a table into a configuration in the chosen target language. FWS can be also used for migrating a firewall policy from a system to another, e.g., from `iptables` to `pf`. Some extra features are given for analyzing configurations:

- *Synthesis*, i.e., the tool prints only the subset of τ-pairs related to some addresses, so reducing the time needed for decompiling when interested only in a portion of the network;
- *Reachability*, i.e., the tool checks whether or not a certain address is reachable from another one, possibly through NAT;
- *Policy implication and equivalence*, i.e., the tool checks if the packets accepted by one configuration are at least/exactly the same accepted by another configuration;
- *Policy difference*, i.e., the tool shows what packets are accepted by one configuration and denied by another. This feature is particularly useful when maintaining a policy to check how updates affect the firewall behavior, because one can see which packets are accepted and which filtered out when a specific rule is added;
- *Related rules*, i.e., the tool list which rules affect the processing of the packets identified by a user-provided query;
- *Policy non-determinism*, i.e., the tool checks if there are packets non-deterministically accepted or dropped.

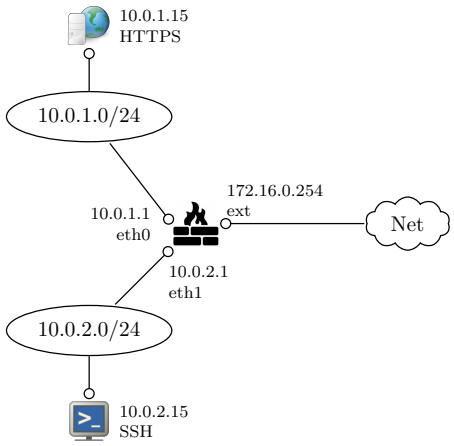


Figure 2.4: A case study of a firewall with three interfaces.

Finally, we show how a system administrator can use FWS to refactor and port a configuration. We describe a typical network of a small company, and we state some requirements that specify the desired security policy. We then consider two firewall configurations in `iptables` and `ipfw`. Finally, we apply FWS to decompile the actual configurations in a tabular, human-readable format and check whether they meet the requirements.

We use the following running example as a scenario for presenting the main features of FWS.

Example 2.17 (Network structure and policy requirements)**.** consider the network shown in Figure 2.4. The `internal` network (10.0.0.0/16) consists of two subnetworks:

- network `lan0` (i.e. 10.0.1.0/24) contains servers and production machines, including a HTTPS server (`web_server`: 10.0.1.15) that runs the company website on port 443;
- network `lan1` (i.e. 10.0.2.0/24) contains the employees's computers, including the system administrator's (`ssh_server`: 10.0.2.15) where a SSH service is running on port 22.

The firewall has three network interfaces: `eth0` connected to `lan0` with IP `lan0_ip` (10.0.1.1); `eth1` connected to `lan1` with IP `lan1_ip` (10.0.2.1); and `ext` connected to the Internet with public IP `ext_ip` (23.1.8.15).

We want to enforce the following requirements on the internal and external traffic:

1. internal networks can freely communicate;
2. connections to the public IP on ports 443 and 22 are translated (DNAT) to `web_server` and `ssh_server`, respectively. This condition permits external hosts to access the website by connecting to the public IP address

```
### NAT rules ###
*nat
# Default policy ACCEPT in nat chains
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

# Req. 2: Redirect incoming SSH and HTTPS connections
# to hosts 10.0.2.15 and 10.0.1.15 (DNAT)
-A PREROUTING -p tcp -d 23.1.8.15 --dport  22 -j DNAT --to 10.0.2.15
-A PREROUTING -p tcp -d 23.1.8.15 --dport 443 -j DNAT --to 10.0.1.15
# Req. 4: Connections towards the Internet exit
# with source address 23.1.8.15 (SNAT)
-A POSTROUTING -s 10.0.0.0/16 ! -d 10.0.0.0/16 -j SNAT --to 23.1.8.15

COMMIT

### Filtering rules ###
*filter
# Default ACCEPT in output (Req. 5), DROP in the other chains
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

# Allow established packets
-A FORWARD -m state --state ESTABLISHED -j ACCEPT
-A INPUT   -m state --state ESTABLISHED -j ACCEPT
# Req. 1: Allow arbitrary traffic between internal networks
-A FORWARD -s 10.0.0.0/16 -d 10.0.0.0/16 -j ACCEPT
# Req. 3: Allow HTTP/HTTPS outgoing traffic
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 80  -j ACCEPT
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 443 -j ACCEPT
# Req. 2: Allow SSH/HTTPS incoming traffic to the corresponding hosts
-A FORWARD -p tcp -d 10.0.2.15 --dport  22 -j ACCEPT
-A FORWARD -p tcp -d 10.0.1.15 --dport 443 -j ACCEPT

COMMIT
```

Figure 2.5: The policy of Example 2.17 in `iptables`.

`ext_ip` at port 443, that is redirected to the corresponding internal host; similarly for accessing the SSH server;

3. connections from the internal hosts to the Internet are allowed only towards HTTP and HTTPS web servers, i.e., with destination ports 80 and 443, respectively;

4. source addresses of packets from the internal hosts to the Internet are translated (SNAT) to the external IP address of the firewall. This allows hosts with private IPs to access the Internet;

5. the firewall can connect to any other host.

Below, we encode the requirements above as queries that are checked by FWS.

Table 2.4: Results of FWS when checking the `iptables` configuration of Figure 2.5; * denotes any value, and $*\backslash S$ denotes any value except for those included in S.

(a) Requirement 1

SrcIP	SrcPort	DstIP	DstPort	Protocol	State
lan0	*	lan1	*	*	NEW
lan1	*	lan0	*	*	NEW

(b) Requirement 2

SrcIP	SrcPort	DstIP	DstPort	DNAT IP	Protocol	State
*	*	ext_ip	22	ssh_server	tcp	NEW
*	*	ext_ip	443	web_server	tcp	NEW

(c) Requirements 3 and 4

SrcIP	SrcPort	SNAT IP	DstIP	DstPort	Protocol	State
internal	*	ext_ip	* \ {internal}	443, 80	tcp	NEW

(d) Requirement 5

SrcIP	SrcPort	DstIP	DstPort	Protocol	State
ext_ip	*	*	*	*	NEW

Compliant configuration in `iptables`

We provide a configuration in `iptables` for Example 2.17. Then, we use FWS to decompile and analyze the configuration and check if it complies with the requirements 1-5 above.

Figure 2.5 shows the policy for our example in the standard `iptables-save` format used to store `iptables` rules in a configuration file. The first sequence of commands delimited by `*nat` and `COMMIT` keywords sets the default policies of all `nat` chains to `ACCEPT`, inserts into the `nat PREROUTING` chain the rules for redirecting the incoming connections to the internal servers (requirement 2) and adds to the `nat POSTROUTING` chain the rule for `SNAT` (requirement 4).

The second block from lines `*filter` to `COMMIT` specifies a default `DROP` policy for the `INPUT` and `FORWARD` chains and a default `ACCEPT` policy for the `OUTPUT` chain, letting the firewall communicate with any host (requirement 5). The first two filtering rules allow the packets belonging to connections flagged as established to go through and towards the firewall, i.e., whenever a new connection is allowed any further packet belonging to the same connection will also be allowed. This is not explicitly required by the policy but is necessary to ensure functionality of connection-oriented protocols. Then, we have `ACCEPT` rules corresponding to the requirements 1, 3 and 2, respectively. Notice that requirement 2 has also rules in the `nat` table above.

We now use FWS to check that the configuration of Figure 2.5 meets the requirements 1-5. First, we load the policy `p` into FWS by specifying the `iptables` rules and a configuration file that encodes what is shown in Figure 2.4. We then ask the tool if arbitrary traffic is allowed among internal networks, raising the

query

```
synthesis(p) in forward/filter where
    ( (srcIp = lan0 and dstIp = lan1) or
    (srcIp = lan1 and dstIp = lan0) ) and state = NEW
```

where `srcIp`, `dstIp` represent the fields for source and destination address of the IP packet entering the firewall interfaces, and `state` tells if a connection is new or established. The query checks whether hosts with `srcIp` in `lan0` can start new connections towards those with `dstIp` in `lan1`, or vice versa, as stated by requirement 1. The operator `=` constrains a variable to be equal to a value or inside a certain interval; the operators `and` and `or` stand for logical conjunction and disjunction.

The output we obtain from the tool is in Table 2.4a, where `*` denotes any value. The table contains all the allowed connections matching the query, confirming that requirement 1 is satisfied. Note that FWS supports the projection of the result to a subset of the available columns using the `project` directive. Hence, from now onwards the results will be projected to the minimum set of columns that represent the considered packets. Moreover, we omit the columns **DNAT IP**, **DNAT Port**, **SNAT IP** and **SNAT Port** when no translation occurs.

We now check that external hosts can access the web and the SSH servers only by connecting to the firewall IP address `ext_ip` at port 443 and 22 respectively (requirement 2). To do that, we ask which packets reach the `web_server` and `ssh_server` hosts:

```
synthesis(p) in forward/nat where
    dnatIp in (web_server, ssh_server) and state = NEW
```

The operator `in` constrains a variable to be equal to a value or inside one of the intervals in the list, hence the notation above is just syntactic sugar for dnatIp = web_server or dnatIp = ssh_server. The variable `dnatIp` represents the destination address of the packet after a DNAT translation. The result in Table 2.4b confirms that requirement 2 is satisfied: indeed, the servers are reachable from any host connecting to the firewall on ports 443 and 22, only.

The next query checks the requirements 3 and 4 together:

```
synthesis(p) in forward where
    srcIp = internal and not (dnatIp = internal) and state = NEW
```

Intuitively, the query asks for the new connections that are allowed from an internal source to an external destination. The answer in Table 2.4c shows that both the requirements are met. Indeed, the notation $* \setminus \{\text{internal}\}$ represents all destination addresses except those in the `internal` subnet. Finally, by checking requirement 5 with the query

```
synthesis(p) in output where srcIp = ext_ip and state = NEW
```

we obtain the output of Table 2.4d showing that the firewall can reach any host.

We can thus conclude that the configuration in Figure 2.5 is correct with respect to the requirements.

```
# NAT setup. The first line defines the SNAT for packets leaving
# the firewall through the interface ext (Req. 4), the other two
# lines specify to perform DNAT on packets arriving
# to the ports 22 and 443 of the firewall (Req. 2)
ipfw -q nat 1 config if ext unreg_only reset redirect_port \
tcp 10.0.1.15:443 443 redirect_port tcp 10.0.2.15:22  22

# Allow established packets
ipfw -q add 0001 check-state
# Req. 1: Allow arbitrary traffic between internal networks
ipfw -q add 0010 allow all from 10.0.0.0/16 to 10.0.0.0/16
# Req. 2: Apply DNAT on packets arriving to the external
# interface of the firewall
ipfw -q add 0100 nat 1 ip from any to 23.1.8.15 in recv ext
# Req. 2: Allow SSH/HTTPS incoming traffic to the corresponding
# hosts and responses from these services
ipfw -q add 0200 allow tcp from any to 10.0.1.15 443
ipfw -q add 0201 skipto 1000 tcp from 10.0.1.15 443 to any
ipfw -q add 0300 allow tcp from any to 10.0.2.15 22
ipfw -q add 0301 skipto 1000 tcp from 10.0.2.15 22 to any
# Req. 3, 4: Allow HTTP/HTTPS outgoing traffic
ipfw -q add 0500 skipto 1000 tcp from 10.0.0.0/16 to any \
80,443 setup keep-state
# Req. 5: Allow arbitrary outgoing traffic by the firewall
ipfw -q add 0501 allow ip from me to any setup keep-state
# Default DROP policy
ipfw -q add 0999 deny all from any to any
# Req. 4: Apply SNAT to outgoing connnections
ipfw -q add 1000 nat 1 ip from any to not 10.0.0.0/16 out
ipfw -q add 1001 allow ip from any to any
```

Figure 2.6: The policy of Example 2.17 in `ipfw`.

Noncompliant configuration in `ipfw`

Figure 2.6 implements the example policy in `ipfw`. On purpose, we introduce subtle but realistic differences with respect to the one in `iptables` and we show how FWS spots them in a clear and concise way.

The first command declares NAT rules, named `nat 1`, that will be activated by the following rules. Note that the next commands have numbers (after the `add` keyword) that can be used for jumps, as we will see below. We refer to those numbers in the description. Command `0001` accepts all the packets that belong to already established connections (command `check-state`). As for `iptables` this is important to ensure functionality of connection-oriented protocols. Command `0010` enables traffic between internal networks (requirement 1). Command `0100` applies `nat 1` to the packets received via the interface `ext`, implementing the DNAT of requirement 2. The actual connections to hosts 10.0.1.15 (`web_server`) and 10.0.2.15 (`ssh_server`), respectively on ports 443 and 22, are enabled by the next commands `0200`, `0201`, `0300` and `0301`. Notice that packets coming from those hosts are handled by jumping (command `skipto 1000`) to the last but one line, which applies `nat 1`, translating source address to 23.1.8.15 (`ext_ip`) (SNAT). Then, packets are accepted by command `1001`. Next line (command `0500`) implements the requirements 3 and 4 similarly to previous rules, i.e., by jumping to `1000` which enforces the SNAT on outgoing

Table 2.5: Results of FWS when checking the `ipfw` configuration of Figure 2.6

(a) Requirement 1

SrcIP	SrcPort	DstIP	DstPort	Protocol	State
lan0	*	lan1	*	*	NEW
lan1	*	lan0	*	*	NEW

(b) Requirement 2

SrcIP	SrcPort	DstIP	DstPort	DNAT IP	Protocol	State
* \ { lan0, lan1, loopback }	*	ext_ip	22	ssh_server	tcp	NEW
* \ { lan0, lan1, loopback }	*	ext_ip	443	web_server	tcp	NEW

(c) Requirements 3 and 4

SrcIP	SrcPort	SNAT IP	DstIP	DstPort	Protocol	State
ssh_server	22	ext_ip	* \ {internal}	*	tcp	NEW
internal	*	ext_ip	* \ {internal}	443, 80	tcp	NEW
web_server	443	ext_ip	* \ {internal}	*	tcp	NEW

(d) Requirement 5

SrcIP	SrcPort	DstIP	DstPort	Protocol	State
ext_ip	*	*	*	*	NEW

connections. Command `keep-state` is the counter-part of `check-state`: the connection is saved in the firewall state so that packets belonging to the same connection will be allowed through the firewall by rule `0001`. Rule `0501` allows the firewall host to communicate to any host. Finally, command `0999` rejects any packet that does not match any previous rule, implementing a default deny policy.

We now use FWS to check if the configuration of Figure 2.6 meets the requirements 1-5 of Example 2.17. We pose exactly the same queries done for the `iptables` configuration in Figure 2.5. In fact, one of the advantages of our approach is that the analysis is fully independent of the particular firewall system and of the language analyzed.

Queries for the requirements 1 and 5 give exactly the same results we got for `iptables` (cf. Table 2.5a, 2.5d and Table 2.4a, 2.4d). Instead, we get an interesting difference while considering requirement 2. For the `ipfw` configuration we obtain that the `web_server` and `ssh_server` hosts cannot be reached by the internal network and by the firewall host via DNAT (cf. Table 2.5b). This is because in the `ipfw` configuration, rule `0100` is defined for interface `ext`, i.e., for packets received from the Internet. In fact, requirement 2 could be interpreted in this stricter way by a system administrator, as hosts `web_server` and `ssh_server` are anyway reachable from internal hosts even without DNAT. FWS spots this subtle difference in the two configurations. To make the `ipfw` configuration behave as the `iptables` one (for requirement 2), it is enough to remove

`recv ext` from rule `0100`.

In checking the requirements 3 and 4, FWS reports that the hosts `web_server` and `ssh_server` can start new connections from source ports 443 and 22, respectively, to any other host. This is due to rules `0201` and `0301` that enable the two hosts to answer connections done through the `DNAT` and provides an alternative way to make connection-oriented protocols work without exploiting the `check-state` command. In principle, this should be considered non-compliant with requirement 3 as new connections from 443 and 22 from the two hosts will access any port and not just 80 and 443, as requested. Again, FWS spots this difference in the policy. This error can be rectified by removing rules `0201` and `0301` and by adding the `keep-state` keyword to the rules `0200` and `0300`.

Interestingly, FWS can compute the equivalence of configurations written for different firewall systems/languages. In this particular case, FWS outputs that the fixed `ipfw` configuration and the `iptables` one are equivalent, as far as the five requirements are in order.

Maintaining firewall configurations

In this section, we show how FWS can be used to maintain the `iptables` policy of Figure 2.5.

Suppose that the company has added a new machine to the `lan0` subnet, which has been assigned the IP address 10.0.1.22. Differently from the other hosts of the network, we want to allow Internet access (with `SNAT`) to this machine only over HTTPS. The other requirements on the traffic should be preserved. For this purpose, we add the following rule to the `FORWARD` chain, which drops connections to port 80 from host 10.0.1.22:

```
-A FORWARD -s 10.0.1.22 -p tcp --dport 80 -j DROP
```

However, we must be careful about the position where to place this rule in order to fulfill the desired requirement and avoid to unintentionally block legal traffic. We discuss below three cases and show how FWS helps in determining the correct position.

The first case is when we place the new rule at the end of the `FORWARD` chain, which has no effect: the *policy equivalence* analysis of FWS reports that the new policy is equivalent to the previous version. To understand the reason why, we use the *related rules* analysis to detect which rules are relevant for processing HTTP packets. The output of the tool only includes the following filtering rule from the `FORWARD` chain:

```
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 80 -j ACCEPT
```

The above rule accepts all the HTTP traffic from the internal networks and is evaluated before the new `DROP` rule. Hence, our new rule should be placed before this one.

The second case is when we insert the new rule before those of the other requirements, e.g., after the rules that allow packets of incoming connections. Now, FWS reports that the policy is not equivalent to the previous one. We check

Table 2.6: Maintenance of the `iptables` configuration. The rules in lines with a + are added, while those with a − are eliminated.

(a) Policy differences after the wrong update

+/-	SrcIP	SrcPort	DstIP	DstPort	Protocol	State
+	internal \ {10.0.1.22}	*	internal	80	tcp	NEW
-	internal	*	internal	80	tcp	NEW

+/-	SrcIP	SrcPort	SNAT IP	DstIP	DstPort	Protocol	State
+	internal \ {10.0.1.22}	*	ext_ip	* \ {internal}	80	tcp	NEW
-	internal	*	ext_ip	* \ {internal}	80	tcp	NEW

(b) Policy differences after the correct update

+/-	SrcIP	SrcPort	SNAT IP	DstIP	DstPort	Protocol	State
+	internal \ {10.0.1.22}	*	ext_ip	* \ {internal}	80	tcp	NEW
-	internal	*	ext_ip	* \ {internal}	80	tcp	NEW

the impact of our changes by running the *policy difference* analysis projected over the HTTP traffic:

```
diff(iptables, updated) in forward where
    protocol = tcp and dstPort = 80
```

The answer to the query is shown in Table 2.6a. The first column is + or - for lines that appear in the synthesis or disappear after the updates, respectively. We see that host 10.0.1.22 is now unable to connect to the Internet, as desired (second table of Table 2.6a). However, our update also prevents communications over HTTP with other machines on the internal networks, thus violating requirement 1 (first table of Table 2.6a).

The correct place where to add the new rule is between the rule for requirement 1 and those for requirement 3. In this way we allow HTTP traffic from 10.0.1.22 to the internal networks, only. If we repeat the check for policy difference, we see that now the only difference is just in the HTTP traffic towards the Internet, as desired (cf. Table 2.6b).

Checking for non-determinism

Now we show how FWS can detect policy non-determinism. Suppose that the `web_server` host has a slow backend so we replicate it for ensuring an acceptable quality of service. The new servers are assigned IP addresses 10.0.1.16 and 10.0.1.17. In this scenario `iptables` can also be used for load balancing at the connection level, by specifying a range of IP addresses in the DNAT target:

```
-A PREROUTING -p tcp -d 23.1.8.15 --dport 443 -j DNAT --to 10.0.1.15-10.0.1.17
```

This rule redirects new connections to a different server in the range following a round-robin discipline. The packets after the DNAT need to be explicitly accepted in the FORWARD chain:

Table 2.7: A packet dropped by a non-deterministic configuration

SrcIP	SrcPort	DstIP	DstPort	DNAT IP	Protocol	State
* \ {internal}	*	ext_ip	443	10.0.1.16	tcp	NEW

Table 2.8: Tests performed on the policy of the Venice CS department (530 rules); times expressed in m:s.cs

Analysis	Multi-cubes	Time
$N_1 \to N_2$	35	0:53.73
$N_1 \to N_3$	28	0:37.77
$N_1 \to Out$	25	1:20.65
$N_2 \to N_1$	45	0:45.32
$N_2 \to N_3$	39	0:34.27
$N_2 \to Out$	31	0:57.40
$N_3 \to N_1$	47	2:19.16
$N_3 \to N_2$	17	0:05.68
$N_3 \to Out$	8	0:09.45
$Out \to N_1$	52	6:02.08
$Out \to N_2$	10	0:11.41
$Out \to N_3$	8	0:08.12
Complete policy	138	17:09.31

```
-A FORWARD -p tcp -d 10.0.1.15-10.0.1.17 --dport 443 -j ACCEPT
```

Suppose now that the host 10.0.1.16 is isolated from the network for maintenance or to mitigate a breach. A naive solution is dropping every connection directed to the server:

```
-I FORWARD 1 -p tcp -d 10.0.1.16 --dport 443 -j DROP
```

However, this rule introduces non-determinism in the firewall behavior: a connection to `ext_ip` on port 443 is dropped if the `DNAT` target is 10.0.1.16, while it is accepted if another server is selected. Our tool identifies these situations and synthesizes the affected packets. The following query performs this check, and the output in Table 2.7 confirms that the examined configuration is non-deterministic:

```
nondet(p) in forward
```

Validation

We have used our tool on several policies to assess how our approach scales to real-world scenarios. We have performed our tests on a desktop PC (running Ubuntu 16.04.2) equipped with an Intel i7-3770 CPU and 16 GB of RAM.

A departmental policy The network of the computer science department of Ca' Foscari is logically partitioned in the main network N_1, the labs network N_2 and a mixed network N_3. The firewall acts as a router between these networks and is connected to the Internet via other routers. The policy is written

in `iptables`, consists of 530 rules (including both `SNAT` and `DNAT`) and contains 5 user-defined chains. In Table 2.8 we report the execution times and the sizes of the obtained specifications when running our tool on the policy projected on specific source and destination networks, as well as the time required to decompile the entire firewall policy. The analysis on specific source and destination networks takes less that one minute most of the times and six minutes in the worst case.

Other real-world policies The authors of [51] have collected a set of anonymized `iptables` configurations from several institutions and from the Internet. Table 2.9 reports the time needed to perform a compilation and decompilation of these policies, together with their size and the number of multi-cubes of the synthesized specification.

Our experiments show that the cost of decompilation increases linearly on the number of rules of the configuration (see Figure 2.7b), and decreases linearly on the number of multi-cubes (see Figure 2.7a). This is expected: obtaining a low number of multi-cubes from a large configuration requires a lot of multicubes extensions, that is performed in line `6` of Algorithm 1 and requires multiple calls to the SMT solver. The cost with respect to both rules and multicubes is shown in Figure 2.8a, where it is clear that the worst case is indeed a large configuration that is decompiled as a small number of multi-cubes, like for `veroneau.net`. A similar rule does not emerge for compilation, see Figure 2.7d, Figure 2.7c and Figure 2.8b. The cost of the compilation is always above 3 seconds in our experiments, also for quite large configurations, thus it is likely that implementation-dependent cost shadows the actual asymptotic cost of the algorithm.

As expected by a compiled code, the configuration obtained may have a weaker structure than the configurations written by hand. Pragmatically, system administrators group the rules in the rulesets according to some usage criterion, also taking into account the features of the network, e.g., its topology, the services it hosts, etc. Yet, the administrators are required some efforts to maintain this structure, while at the moment our tool has no information for keeping such a structure. Furthermore, the compilation of a multi-cube may result in multiple rules, because most of the target languages cannot directly express conditions on sets of ranges of addresses. For this reason, the transcompiled configuration may be longer than the original one. However, FWS is intended to provide its users with means for understanding and checking a configuration in its synthesized version through the provided query language, rather than inspecting the target configuration as it is.

The repository also contains the firewall configuration of the lab the authors of [51] are affiliated to. The firewall has 22 network interfaces and its policy consists of 4841 `iptables` rules. In Table 2.10 we provide a summary of the time required to produce a synthesis of the policy with no checks on MAC addresses for each possible pair of input/output interfaces and to communicate with the Internet. Most of the analyses terminate in less than 3 minutes and just a couple

Table 2.9: Tests performed on real-world policies; times expressed in m:s.ms

Description	Rules	Multi-cubes	Decompilation	Compilation
Policy from Github	15	11	00:00.765	00:00.072
Ticket from OpenWRT	65	11	00:01.519	00:00.029
Kerberos server	8	14	00:01.635	00:00.111
Policy from a blog	28	25	00:02.572	00:02.092
Eduroam laptop	21	15	00:01.018	00:00.061
Memphis testbed	34	15	00:01.233	00:00.049
Kornwall	52	23	00:02.362	00:00.067
Shorewall	77	48	00:28.154	00:02.398
Home router	76	36	00:05.879	00:02.783
Medium-sized company	90	20	00:25.289	00:00.397
`veroneau.net`	263	7	05:55.690	00:01.696

Table 2.10: Tests performed on the *Chair for Network Architectures and Services* firewall policy

Analysis	<1m	1-3m	3-5m	5-10m	10-20m
Subnet → Subnet	0	405	37	20	0
Subnet → Internet	14	5	1	1	0
Internet → Subnet	5	13	1	0	2

of cases involving particularly complex subnets take more than 10 minutes to be completed. In these particular cases, the time for the synthesis increases, because the policy contains thousands of rules mapping IP to MAC addresses. In fact, a typical use case is to check that these bindings are correctly enforced. This can be achieved through queries on singles IPs which complete in less than 2 minutes.

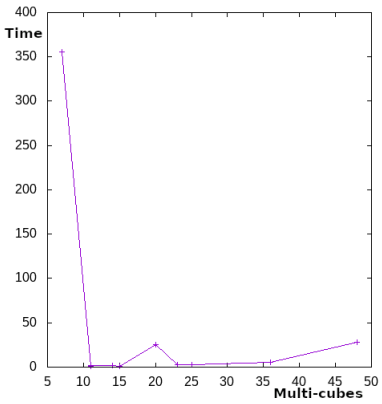
We have performed some tests to evaluate the expressiveness of the output produced by FWS. For instance, in the *Home router* example, we can check which hosts in the private LAN are reachable via the public IP address of the router by running the query

```
dstIp == 117.195.222.105 && state == NEW
```

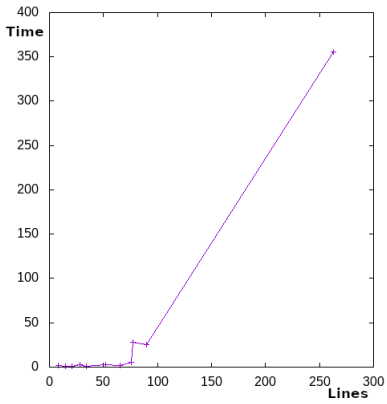
FWS succinctly reports that external hosts can access the internal server 192.168.1.130 on ports 22, 80, 443 and 1194 via `DNAT`. For hosts in the private LAN 192.168.1.0/24, both `SNAT` and `DNAT` are applied to connections towards the public IP address to avoid the problem of asymmetric routing (also known as NAT reflection). For lack of space, we do not discuss the remaining examples that are available online [121].

2.7.2 F2F

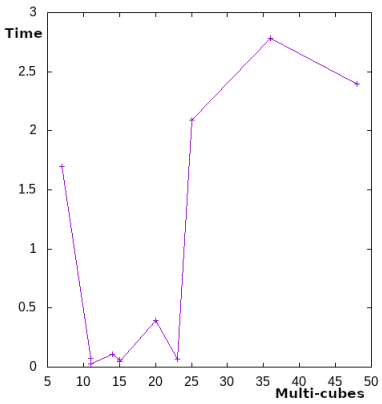
We implement a tool, called *F2F* [12] that applies Algorithm 4. To support the user in analyzing and porting real-world policies, the tool gets as input a policy expressed in one of the configuration languages of subsection 2.2.2 or as a fw-function in a tabular form. Also, the user chooses the wanted target language. When the user provides a configuration, the tool computes its semantics as a



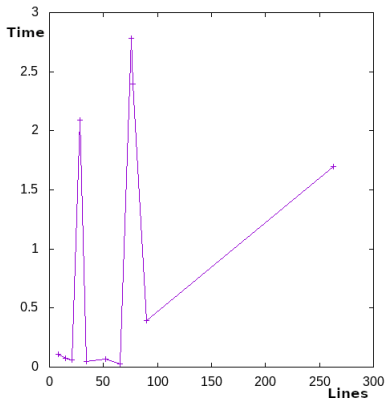
(a) Decompilation time over multi-cubes.



(b) Decompilation time over rules number.



(c) Compilation time over multi-cubes.



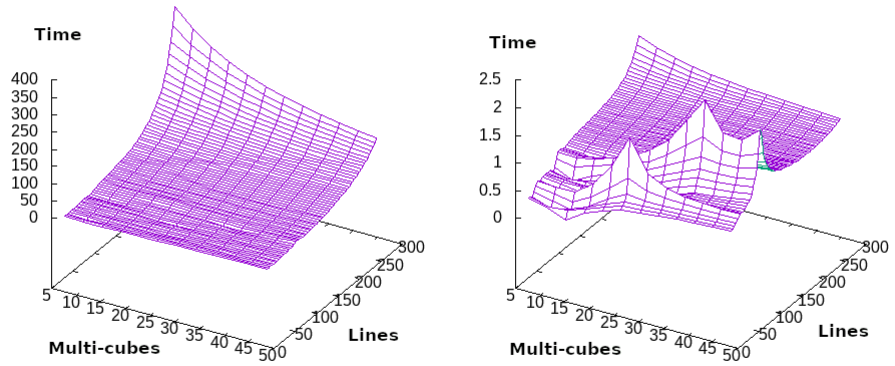
(d) Compilation time over rules number.

Figure 2.7: Time cost of decompilation and compilation policies of Table 2.9.

preprocessing step, thus obtaining a fw-function. Then, it checks if the policy is expressible by the target language, and notifies the user with exact information if this is not the case.

Figure 2.9 sketches the workflow of *F2F* in the case of porting a configuration from `ipfw` to `pf`. In the top part there is the source configuration that is then encoded in IFCL, from which the tool extracts the semantics as a fw-function. The bottom part depicts the control diagram and the label assignment for `pf`, and the step computing its expressivity. The tool than checks whether the semantics of the source configuration is expressible in `pf`, and if this is not the case, it produces a report with the inexpressible and clashing pairs.

For the computation of the semantics we rely on a tailored version of *FWS*.



(a) Decompilation time.

(b) Compilation time.

Figure 2.8: Decompilation and Compilation time over rules and multi-cubes.

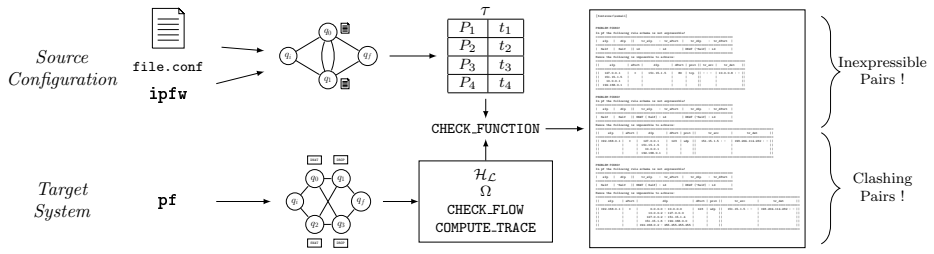


Figure 2.9: Schema of *F2F*.

Validation on a Case Study: FirewallBuilder

Consider again the scenario in Figure 2.3 showing a typical network of a small company, and assume the administrator produces a configuration for `pf` using the policy management system *FirewallBuilder*. Below we show how *F2F* may fit a policy creation and management workflow, helps the administrator in understanding the limitations of FirewallBuilder and of `pf`, and supports possible fixes of the detected problems. We also consider `iptables` and `ipfw` and we show that they have similar weaknesses.

FirewallBuilder FirewallBuilder is a well-known tool for Unix-based systems that supports the administrator to write firewall policies in a tabular form and then compiles them to the most common firewall languages. Two separate tables are given, the first one for network translation and the second one for packet

```
$ sudo ./f2f table Example/interfaces Example/table.conf pf

!!! Inexpressible Pair Found !!!
=================================================================================================
||     sIp     | sPort |              dIp              | dPort | prot ||      tr_src     |    tr_dst    ||
=================================================================================================
|| 192.168.0.1 |   *   |      0.0.0.0 - 151.15.1.4     |   53  |  *   || 151.15.1.5 : id | 8.8.8.8 : id ||
||             |       |  151.15.1.6 - 192.167.255.255 |       |      ||                 |              ||
||             |       | 192.168.1.0 - 255.255.255.255 |       |      ||                 |              ||
=================================================================================================

!!! Inexpressible Pair Found !!!
=============================================================================================
||    sIp     | sPort |              dIp              | dPort | prot ||  tr_src |    tr_dst    ||
=============================================================================================
|| 151.15.1.5 |   *   |      0.0.0.0 - 151.15.1.4     |   53  |  *   || id : id | 8.8.8.8 : id ||
||            |       |  151.15.1.6 - 192.167.255.255 |       |      ||         |              ||
||            |       | 192.168.1.0 - 255.255.255.255 |       |      ||         |              ||
=============================================================================================

!!! Clashing Pairs Found !!!
(P1, t1):
=============================================================================
||              sIp              | sPort |     dIp     | dPort | prot ||  tr  ||
=============================================================================
||      0.0.0.0 - 151.15.1.4     |   *   | 192.168.0.7 |  443  |  *   || DROP ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||
=============================================================================
(P2, t2):
=============================================================================================
||              sIp              | sPort |    dIp     | dPort | prot ||  tr_src |      tr_dst      ||
=============================================================================================
||      0.0.0.0 - 151.15.1.4     |   *   | 151.15.1.5 |  443  |  *   || id : id | 192.168.0.7 : id ||
||  151.15.1.6 - 192.167.255.255 |       |            |       |      ||         |                  ||
|| 192.168.1.0 - 255.255.255.255 |       |            |       |      ||         |                  ||
=============================================================================================
in node q3:
with [P@ || t1@ || t2@]:
=============================================================================================
||              sIp              | sPort |     dIp     | dPort | prot || tr1  || tr2_src | tr2_dst ||
=============================================================================================
||      0.0.0.0 - 151.15.1.4     |   *   | 192.168.0.7 |  443  |  *   || DROP || id : id | id : id ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||         |         ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||         |         ||
=============================================================================================
Hint: Apply tags to P1 in node q2 and use them to choose the transformation in node q3

!!! Clashing Pairs Found !!!
(P1, t1):
=============================================================================
||              sIp              | sPort |     dIp     | dPort | prot ||  tr  ||
=============================================================================
||      0.0.0.0 - 151.15.1.4     |   *   | 192.168.0.6 |   22  |  *   || DROP ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||
=============================================================================
(P2, t2):
=============================================================================================
||              sIp              | sPort |    dIp     | dPort | prot ||  tr_src |      tr_dst      ||
=============================================================================================
||      0.0.0.0 - 151.15.1.4     |   *   | 151.15.1.5 |   22  |  *   || id : id | 192.168.0.6 : id ||
||  151.15.1.6 - 192.167.255.255 |       |            |       |      ||         |                  ||
|| 192.168.1.0 - 255.255.255.255 |       |            |       |      ||         |                  ||
=============================================================================================
in node q3:
with [P@ || t1@ || t2@]:
=============================================================================================
||              sIp              | sPort |     dIp     | dPort | prot || tr1  || tr2_src | tr2_dst ||
=============================================================================================
||      0.0.0.0 - 151.15.1.4     |   *   | 192.168.0.6 |   22  |  *   || DROP || id : id | id : id ||
||  151.15.1.6 - 192.167.255.255 |       |             |       |      ||      ||         |         ||
|| 192.168.1.0 - 255.255.255.255 |       |             |       |      ||      ||         |         ||
=============================================================================================
Hint: Apply tags to P1 in node q2 and use them to choose the transformation in node q3
```

Figure 2.10: *F2F* output when checking the example policy for `pf`.

Table 2.11: An example policy for a typical network.

dstIP	dstPort	srcIP	srcPort	DNAT	SNAT
not (151.14.1.5 192.168.0.0/24)	53	192.168.0.0/24	any	8.8.8.8 : id	151.15.1.5 : id
not (151.14.1.5 or 192.168.0.0/24)	53	151.15.1.5	any	8.8.8.8 : id	id : id
not (151.14.1.5 or 192.168.0.0/24)	not 53	151.15.1.5, 192.168.0.0/24	any	id : id	151.15.1.5: id
192.168.0.1, 151.15.1.5	22	192.168.0.2 - 192.168.0.255	any	id : id	id : id
192.168.0.2 - 192.168.0.255	any	192.168.0.2 - 192.168.0.255	any	id : id	id : id
151.15.1.5	443	not (151.14.1.5 or 192.168.0.0/24)	any	192.168.0.7 : id	id : id
151.15.1.5	22	not (151.14.1.5 or 192.168.0.0/24)	any	192.168.0.6 : id	id : id

```
rdr from <NotCompany> to 151.15.1.5 port 443 -> 192.168.0.7
rdr from <NotCompany> to 151.15.1.5 port 22 -> 192.168.0.6
nat from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany> port 54 -> 151.15.1.5
nat from { 151.15.1.5 , 192.168.0.0/24 } to { 6.6.6.6, 8.8.8.8 } -> 151.15.1.5
nat from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany> -> 151.15.1.5
nat from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany> port 54 -> 151.15.1.5
rdr from 192.168.0.0/24 to <NotCompany> port 53 -> 8.8.8.8
nat from 192.168.0.0/24 to 8.8.8.8 port 53 -> 151.15.1.5
rdr from 151.15.1.5 to <NotCompany> port 53 -> 8.8.8.8

pass quick from <NotCompany> to 192.168.0.7 port 443
pass quick from <NotCompany> to 192.168.0.6 port 22
pass quick from { 151.15.1.5 , 192.168.0.0/24 } to <NotCompany>
pass quick from 192.168.0.2 - 192.168.0.255 to 192.168.0.2 - 192.168.0.255
pass quick from 192.168.0.2 - 192.168.0.255
    to { 151.15.1.5 , 192.168.0.1 } port 22
block quick  from any  to any
```

Figure 2.11: FirewallBuilder output when compiling the example policy for `pf`, where `<NotCompany>` is a table containing all the IP addresses that are not used by the company.

filtering. Since FirewallBuilder does not come with a clear definition of its semantics, the user not always knows how these tables are used to determine the destiny of packets. Also, the tables may contain rules that are superfluous or conflicting depending on which table is used first, e.g., when a packet p is transformed with t in the translation table, and p or $t(p)$ is discarded in the filtering table. This may lead to clashes when producing the target configuration.

Policy Requirements and Specification We start by putting forward the requirements that firewall policy of the scenario of Figure 2.3 must meet:

1. LAN hosts freely communicate with each other;
2. LAN hosts access the firewall via SSH (port 22), only;
3. The company hosts (a LAN host or the firewall) can freely send packets to the Internet;
4. Packets from the Internet are discarded, if not directed to the public IP of the firewall with port 22 or 443;
5. Packets from the Internet directed to port 22 or 443 are redirected to the internal SSH server (at address 192.168.0.6) or to the HTTPS server (at address 192.168.0.7);
6. When a company host tries to connect to a DNS service on the Internet, on port 53, the packet is redirected to 8.8.8.8;
7. The source address of all the packets leaving a company host towards the Internet is replaced with the public IP of the firewall.

These requirements originate the policy represented in Table 2.11, expressed in a declarative form as the list of the accepted packets and their transformations.

Implementing the Policy and Detecting Problems Typically, defining a policy using FirewallBuilder is straightforward but there may be some cases where the administrator needs to guess how the translation and filtering tables are used to manage packets. For example, consider the last line of Table 2.11 that conflicts with the implicit indication to discard the packets from the Internet and with destination 192.168.0.6. If the administrator assumes that the translation table is checked before the filtering one, the packets directed to 192.168.0.6 must be accepted, thus obtaining the `pf` configuration in Figure 2.11 (which is actually the output of the tool, with a little of maquillage for legibility). However, the configuration fails in encoding the desired policy since Requirement (4) is not met. If instead the filtering table is inspected first, the last line of the policy is simply ignored. Summing up, in neither cases FirewallBuilder implements correctly the policy in Table 2.11, and in addition no warning is notified.

F2F instead signals a clash when running on the considered configuration. The output in Figure 2.10 shows that in node q_3 the packets from the Internet

directed to the internal server in 192.168.0.6 are indistinguishable from the ones originally directed to the firewall because of the DNAT in node q_2. Our tool suggest the administrator to use tags in the node q_2 for distinguishing the two clashing (multi-cubes of) packets. As a matter of fact, there are two clashes, and the tool suggests tags to fix both.

F2F signals other problems in the analyzed configuration, because there are also two sets of inexpressible pairs. The first set is (P, t), where P contains the packets from the external interface of the firewall towards the Internet on port 53, and t is a DNAT to the address 8.8.8.8. In other words, the firewall can access any DNS server on the Internet, not only the one prescribed by the policy. Even though the inexpressible pairs are considered by FirewallBuilder when producing its output configuration, the resulting rules are simply ignored by `pf`. Note that the administrator gets no warning about this problem and could wrongly think that the configuration enforces them (as it seems at first sight). Of course, this misconfiguration is possibly dangerous for security. Moreover, `pf` cannot apply the translations t above to the packets in P, thus the administrator can only fix this problem resorting to external tools. The second set of inexpressible pairs is quite similar to what just described.

Problems with `ipfw` and `iptables` Similar problems arise if one compiles the policy in Table 2.11 to `ipfw` and to `iptables`. More in details, the generated `ipfw` configuration suffers from no clashes but from the same inexpressible pairs. Whereas, the `iptables` configuration presents the same clashes, but no inexpressible pairs.

Performance on real configurations

We evaluated the *F2F* effectiveness against real world configurations [49]. The experiments are performed on a desktop computer with an i7-7700 processor (3.60GHz) and 8Gb RAM, running Ubuntu 20.04.3 LTS. The results are in Table 2.12: the first column reports the name of the configuration; the second one the number of lines of the configuration; the third one the time taken by *F2F* to compute the IFCL-configuration, to extract its fw-function, and to check both kinds of expressivity; finally the last one is the time for checking the expressivity only. Performance is acceptable for all the configurations, and the time for checking expressivity is negligible. When checking `pf` and `ipfw`, we found two inexpressible pairs in the configuration `eduroam_laptop`: in both pairs a DNAT is applied to a packet from an address in $\mathcal{S}$ to one not in $\mathcal{S}$ (lines 11 and 12 of Table 2.3). For the same systems, we also found two clashes in the configuration `medium sized company`, between packets to be both translated and dropped.

2.8 Related Work

The literature has many proposals for simplifying and analyzing firewall configurations. Some are based on formal methods, others consist of *ad hoc* configura-

Table 2.12: Experimental results of *F2F* against real-world configurations.

Configuration	**Lines**	**Total time (*s*)**	**Checking time (*s*)**
ticket_openwrt	128	7.00	0.05
sqrl_shorewall	106	190.10	0.48
random_srv	16	7.43	0.05
memphis_testbed	46	6.51	0.05
medium sized company	639	75.54	0.20
kornwall	88	54.73	0.54
home router	130	20.67	0.21
github_myiptables	53	5.35	0.03
eduroam_laptop	57	7.97	0.10
blog_a	51	11.03	0.13

tion and analysis tools (we only consider here the publicly available ones). These works can be divided depending on the kind of approach they take. Some take a top-down approach, proposing ways to specify abstract filtering policies that can be possibly compiled into the actual firewall systems, or checked against them.

Many works take a top-down approach, proposing ways to specify abstract filtering policies that can be possibly compiled into the actual firewall systems, e.g., [14, 15, 45, 61, 8, 55, 19]. Lots of these approaches only consider the simplest types of rules [61, 55, 19]. NAT, stateful policies and tags are often totally or partially ignored by formal tools.

Other papers take a bottom-up approach and adopt formal methods. To the best of our knowledge, ours is the only one providing at the same time: (*i*) a language for analyzing multiple firewall systems; (*ii*) an effective technique for synthesizing abstract policies; (*iii*) a support for NAT; (*iv*) a formal characterization of firewall behavior. Some researchers focused on analyzing `iptables`: Jeffrey et al. introduce in [79] a formal model of firewall policy, based on `iptables`, and investigate the properties of reachability and cyclicity of firewall policy configurations. The proposal by Diekmann et al. [50] has some similarities with ours. In particular, the authors provide a "cleaned" ruleset that an automatic tool can easily analyze, using a formal semantics of `iptables` mechanized in Isabelle/HOL [98]. Furthermore, they propose a semantics-preserving ruleset simplification (e.g., chain unfolding) with a treatment of unknown match conditions, due to a ternary logic. The subset of `iptables` they consider includes only filtering and access control flow actions, but not packet modification such as NAT. Differently from theirs, our approach supports NAT, packet tagging, is able to detect non-determinism in policies, and it is based on a generic language that can target languages different from `iptables`. The tool ITVal [86] parses `iptables` rules and supports SQL-like queries to discover host reachability. Differently from our FWS, ITVal is specific for `iptables` and aims at answering reachability queries, only. In particular, it neither synthesizes an abstract firewall specification nor ports configurations to different languages and it does not detect non-determinism.

Other proposals in the literature are more general and target, in principle,

various firewall systems. Below, we discuss the main differences with respect to our work. A model-driven approach is proposed in [101] to derive network access-control policies from real firewall configuration. A proof of concept is given only for `iptables`. Moreover, compared to our proposal this paper does not address NAT. In [44] the authors propose an algorithmic solution to detect and correct specific anomalies on stateful firewalls. However, the proposed approach does not aim at synthesizing an abstract specification, as we do. The tool FIREMAN [131] detects inconsistencies and inefficiencies of firewall policies. It does not support NAT though. In [96] the Margrave policy analyzer is applied to the analysis of IOS firewalls. The approach is rather general and extensible to other languages, but the analysis focuses on finding specific problems in policies rather then synthesizing a high-level policy specification. A framework for the static analysis of networks is proposed in [81]. It provides sophisticated insights about network configurations but does not specifically analyze real firewall configurations and, as for the previous papers, there is no synthesis of high-level specifications. Fang [87] is another tool for querying real policies in order to discover anomalies. Its authors state that it synthesizes an abstract policy that resembles the one we propose here, but we have been unable to use the tool and the paper does not describe its internals, making any comparison with our approach impossible.

Ranathunga et al. [105] introduce an algebraic description of firewall configurations where rules are monoid-endomorphisms transforming sequences of packets. Rulesets are obtained by suitably combining these endomorphisms. The authors exploit this representation to define algorithms for policy checking, implication and differences, which are implemented in the Malachite tool. Differently from ours, their proposal does not seem to automatically generate the algebraic representation from real configurations. Furthermore, Malachite seems only to analyze the policy without generating back a clean configuration.

Hallahan et al. [67] propose Firemason, a tool that verifies a firewall configuration against a given specification, producing a counterexample if it does not. When this happens, Firemason can also synthesize a fix using behavioral examples provided by users that describe the correct behavior. Similar to our proposal, Firemason internally converts a configuration into a logical formula and uses Z3 to perform the verification step and to generate the fix for the configuration. Differently from FWS, Firemason can handle rate limit rules, but currently it supports only `iptables`, and therefore offers no features to automatically port configurations. Finally, it does not implement any analysis to spot possibly non-deterministic behavior.

Another approach to modeling and verifying firewall policies consists in using Binary Decision Diagrams to efficiently represent packet filters, as first proposed by Hazelhurst in [68, 69]. These data structures concisely represent the boolean expressions that describe which packets must be accepted and which rejected. A tool is also provided that can analyze rule sets. Differently from ours, this approach mainly focusses on Cisco and does not address NAT issues.

Other papers study the definition of network configurations and in particular their verification. Note that this is a different issue than ours, which instead

focusses on well-established systems and languages. Note also that these two approaches can be combined together.

Anderson et al. [18] introduced NetKAT, a language for programming a collection of network switches. It is equipped with a denotational semantics and an axiomatic one, both based on Kleene algebra with tests. Although the primitives provided by NetKAT are similar to those of IFCL, i.e., for filtering, modifying, and transmitting packets, its focus is different from ours.

Fogel et al. [54] proposed Batfish, a tool for statically analyzing network configurations. Batfish encodes a configuration and the relevant information of that network, e.g., the used protocols and the data plane, into Datalog. Network administrators can exploit the Datalog deduction machinery to check correctness properties expressed as a first-order-logic formula. If a violation occurs, Batfish produces a counterexample in the form of concrete offending packets.

Bringhenti et al. [28] introduced VEREFOO, a tool that given a set of security requirements and a graph describing the network services, computes two pieces of information. The first one describes where to deploy the security function nodes (i.e., nodes that implement security checks on the traffic flow) in the network to meet the desired security goals. The second output is the configuration that implements the policy for each security function.

Valenza et al. [127] proposed a language-independent approach for the verification and the detection of anomalies of forwarding polices in a SDN scenario. Their proposal can be used both in a top-down or in bottom-up manner. In the first case, network administrators define a policy abstractly and translate it into a configuration to be deployed on a target device. The idea is to use a logical formalism for modeling the behavior of the forwarding policies and the network in hand, and to use a SAT solver for carrying out analyses on this logical model. If no anomaly is found, the forwarding policies are translated into a real SDN language. In the bottom-up approach administrators verify whether a change in the configuration of a network node causes some anomalies, by checking the logical formulation.

We remark that the goal of the above appoaches is modeling an entire network leaving out the details of the single firewalls, expressed in the most used configuration languages, that instead is our target. We see no particular difficulties in applying our proposal to manage the firewalls associated with each node of a network, so integrating the two approaches.

To the best of our knowledge, we are the first to propose a bidirectional approach, in which the high and low levels are bound by compilation and decompilation functions. This is also the first work that formally investigates the expressive power of firewall systems by using programming language-based techniques, and propose algorithmic means to check that a given policy is expressible by the target firewall system.

2.9 Conclusions and Future Work

We have presented an approach for interacting with firewalls at different levels of abstraction for different tasks. Our approach is based on a double representation of the firewall behaviour. One is the low-level executable configuration, and the other one is an abstract representation of the policy, i.e., the emergent behaviour expressed as a function over packets. In particular, we have considered `iptables`, `ipfw` and `pf`, the main firewall systems used in Linux, FreeBSD, OpenBSD and MacOS. We formally modeled both the layers, and gave compilation and decompilation functions for keeping aligned two different representation of a firewall system. We also investigated the expressive power of the considered languages, namely, `iptables`, `pf` and `ipfw`, showing two hierarchy, one that consider tag systems, the other focusing only on the basic and most used features. We gave an algorithm for checking if a given policy can be implemented in the target system that also highlight if tags are needed and how to use them.

We implemented the translation functions and the expressivity checking in a couple of tools that we tested on real-world configurations, showing that their performance are acceptable.

Future Work Future work includes considering different firewall systems, like Cisco-IOS, which is particularly challenging because the control diagram is also affected by routing choices. A promising line of research is about incrementality and compositionality, i.e., only propagates the modification from high to low level representations, without recompiling the whole policy. This would allow to maintain properties of the low level configuration, e.g. logs and the internal structure of the rulesets. Also, we plan to enrich IFCL with features for network interfaces and routing. Tools exist that generate and distribute over the nodes of a network the specification of local policies starting from a global one, e.g., [28]. The local policies obtained by [28] are represented in a tabular form, similar to ours. It would be then easy to compile these local specification in a chosen firewall language. We will further extend our approach to the SDN paradigm. The major difficulty in this case arises because of the high dynamicity of SDNs, while our proposal focuses on legacy networks and devices that are essentially static. Finally, it would be very interesting to extend our approach to deal with networks with more than one firewall. The idea would be to combine the synthesized specifications based on network topology and routing.

Chapter 3

System

In this chapter, we target Operating Systems, focusing on SELinux configurations. 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 [130] network appliances [62], and mobile devices [114]. Defining a 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 serious misconfigurations that have led to vulnerabilities in widely used policies [76].

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 [116, 75]. In particular, SELinux developers recently proposed the intermediate configuration language CIL (Common Intermediate Language). CIL is a promising 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.

Requirements for a system access control commonly predicates on permitted and denied information flow between OS entities [65, 48]. We propose *IFL* (Information Flow Language) a domain specific language (DSL) that can express common fine-grained information flow requirements, including confidentiality, integrity, and non-transitive properties. We group information flow requirements 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 op-

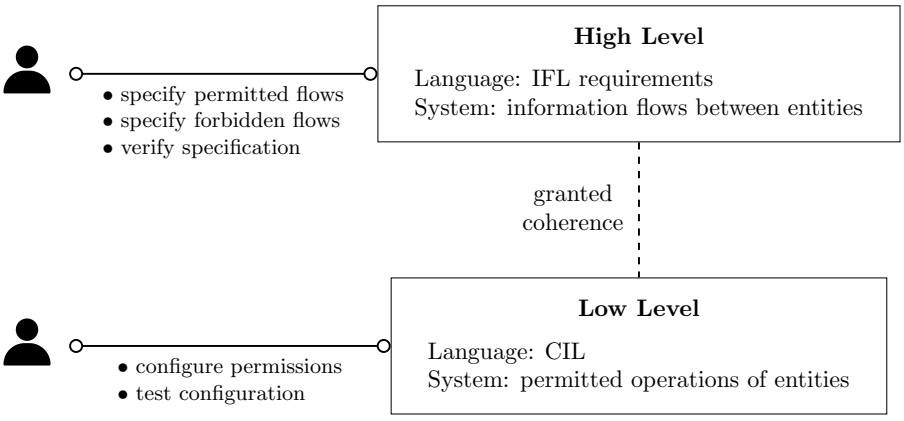


Figure 3.1: Schema of the two-layer approach for SELinux.

erations. In contrast, security requirements prevent entities from operating on other possibly critical entities, and thereby to enforce security properties, including confidentiality, integrity, and non-transitive information flow properties. Given the complexity of modern operating systems, we embedded IFL with a refinement relation that allows the administrator to structure the properties by defining them in a more general way and adding context-dependent details later. IFL is completely integrated in the CIL code, information flow requirements can appear as special comments that an administrator can associate with different parts of a CIL configuration.

The general schema is depicted in Figure 3.1. We identify the high level of the system with information flow specification, abstractly representing the expected behaviour of the system with functional and security requirements. The low level is the one of CIL configurations, where every single permission is listed for each entity. Summarizing, the two abstraction layers are:

- the low level (LL), the one of CIL configurations, where each entity of the operating system is associated with the actions it can perform on other entities;
- the high level (HL), the one of IFL specifications, where requirements states functional and security properties expressed as permitted and forbidden information flows.

The binding between the two layers is obtained by IFCIL, an extension of CIL supporting information flow requirements, and a verification procedure for statically checking that a configuration satisfies its requirements. Intuitively, the languages of the two layers are merged into a new one that shares the same advanced features of CIL and where IFL requirements are first class citizens. The different layers are adequate for performing specific tasks. The administrator interacts either with the system specification by operating on the IFL require-

ment, or with the executable CIL configuration, and the verification procedure ensures that the two layers are coherent.

In summary, our main contributions are as follows.

- We present the language IFL for expressing fine-grained information flow requirements, including confidentiality, integrity, and non-transitive information flow properties. IFL is declarative, compositional, and it allows administrators to specify complex requirements using refinement.
- 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. This algorithm is based on an encoding from IFL into linear-time temporal logic supporting the use of off-the-shelf model checkers for verification.
- We give CIL a formal semantics and empirically validate its adequacy with respect to the CIL reference manual and the CIL compiler. Our experiments highlight some unspecified corner cases and disagreements between the documentation and the compiler.
- We provide a prototype tool [5] that implements our verification procedure by exploiting an existing model checker [40]. Our tool checks if a 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 [62, 63, 64]. 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 thirty nine requirements on the configuration in [62], which has roughly forty six thousands lines of code.

Structure of the chapter

In Section 3.1 we introduce SELinux, CIL, and the mechanism used by administrators to protect critical resources. In Section 3.2 we give a high-level account of our CIL semantics and how we experimentally validate its adequacy. In Section 3.3 we present IFL and IFCIL. In Section 3.4 we explain our verification procedure for checking the satisfaction of the requirements, and we present our verification tool and our experimental assessment. In Section 3.5 we compare our work with the relevant literature and in Section 3.6 we draw conclusions. Appendix B contains the details of our formal development and the proofs of our theorems.

3.1 Background

SELinux SELinux is a set of extensions to the Linux kernel and utilities. 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 such a configuration, an administrator defines a set of *types*, and labels the OS resources and processes with them. In addition, all the resources belong to pre-defined *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*. 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 [76]. Some high-level configuration languages have been suggested with their own compilers and tools as an attempt to address these limitations. Recently, the SELinux developers proposed a promising new intermediate configuration language with advanced features and tools to support both the development of high-level languages and the definition of configurations. We briefly survey this language below.

CIL The Common Intermediate Language (CIL) 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 offer cross-language interaction. A compiler for the kernel policy language is currently available, and CIL is designed to support existing high-level configuration languages [75], and future ones too. Despite its original goal, CIL is also used to directly write configurations [64, 62, 63], for complex real-world policies, e.g., that for Android [59]. 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*, that 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 re-use 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 reusability 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 of the inherited block are made available to 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 previous 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 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 unconfined)
    (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 house.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 add_dog.mcr(cat cat)))
```

Roughly, the content of `add_dog` replaces the last line where the formal parameters `x` and `y` are both bound to `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`. Sometimes name resolution is 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 3.2 for examples), and lead to difficult to spot misconfigurations. 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 the Appendix, providing an intuition in Section 3.2.

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 through `neverallow` rules. The rule below prohibits subjects with type `animal_house.cat` to read resources of any type that is 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 notify an error because `animal_house.cat` can read the files of type `animal_house.man` that is in `not_pet`. However, we argue below that these checks are insufficient to prevent insecure information flow.

An example from a real-world configuration Consider the following block `mem` defined in [62], 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 a 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.

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 an 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 has meaning for our verification tool, but is completely ignored by the standard CIL compiler. Thus, an IFCIL configuration is still a CIL configuration.

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

3.2 Formalizing the Low Level

The official CIL documentation [66] formally describes neither the CIL syntax nor its 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:

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

Note that 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.

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 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 erroneous according to the compiler's behaviour although against it is one's intuition. Indeed, 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. Actually, we discovered that `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. As we will later see, our tool warns the administrator about such misconfigurations and approximates the configuration behaviour by pruning the recursion tree.

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 such semantics focuses on the type enforcement fragment of the language, which is its most used part (see the real-word CIL configurations in Section 3.4.1), 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 in Section B.1.

The semantics benefits from a kind of 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 different 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 the right one below.

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

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

Adequacy of the formalization The adequacy of our formalization is assessed against the available documentation and the CIL compiler. Namely, the equations defining our semantics are consistent with the reference manual, for those cases covered by the documentation, and where the documentation is unambiguous; otherwise our formalization conforms with the compiler. When the documentation and the compiler disagree, and when unexplainable behaviour arises, we have contacted the CIL developers to disambiguate the intended meaning. To obtain test cases we proceeded as follows.

We studied the reference manual and formalized CIL's constructs following their intuitive description. We identified name resolution as the most involved process of the compilation from CIL to the kernel binary representation. For this reason, we generated test cases for each construct in isolation with a special

focus on name resolution. Our objective was both to check the validity of the documentation, and thus of the model we made using it, and to explore the behaviour in the corner cases that were not considered by the reference manual. Actually, we found that some of these test cases do not behave coherently with what is described in the reference manual. In particular, 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 documentation, types defined inside the macro should be checked before the ones defined in the namespace where the macro is defined. Hence, when copying the `allow` rule from `m` to `B`, we expect `a` to be resolved as `B.a`. Instead it is resolved as `A.a`.

We have notified the developers about these cases. They agreed that these are not handled consistently with the expected behavior and that they will address this in a future release [33, 34, 32].

A major concern in CIL is how macro resolution and block inheritance interact, in particular the order in which they are handled. This is unspecified in the documentation and can affect the semantics. Indeed, as showed at the beginning of this section, there are configurations that behave differently depending on the evaluation order of their constructs. The results indicate that macro calls are handled after block inheritance, which in turn are handled independently of each other.

Moreover the various steps needed for handling each construct also affects the target configuration. An example is the evaluation of the parameters of a macro with respect to the copy of its content.

More precisely, we have designed test configurations with various combinations of the considered constructs, whose semantics depends on the order in which their steps are performed. We have considered all the possible orderings, including multiple occurrences of the same step for different occurrences of the same construct.

3.3 Extending CIL with Information Flows

This section introduces IFL, our DSL for defining annotations that enable administrators to express information 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.

3.3.1 The High Level: 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.

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 if `+[o]>`. Nodes may also be given using the wildcard `*` standing for any node representing a type. The non-empty set $o \subseteq O$ contains a subset of the applicable operations, omitted when it is the entire set O. The labeled path $P_1\ P_2$ is additionally constrained so that the ending point of P_1 matches the starting one of P_2.

The direction of arrows reflects how information flows in the graph, e.g., `n [write,read]>` $\texttt{n}'$ 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 `n >` $\texttt{n}'$, whereas an indirect information flow is represented by multiple steps `n +>` $\texttt{n}'$. A kind can also mention intermediate steps, e.g., `n > * >` $\texttt{n}''$ `+>` $\texttt{n}'$ specifies that information flows in two steps (through an unspecified node) from `n` to $\texttt{n}''$ and then in many 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 $\emptyset \neq o' \subseteq o$ are the operations of n on n' that cause an explicit information flow from n to n' (e.g., `write`); (ii) the arc (n', o'', n), where $\emptyset \neq o'' \subseteq o$ are the operations of n on n' that cause an explicit information flow from n' to n (e.g., `read`).

The administrator can state the following requirements on configurations

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

which make assertions about the information flow diagram I and flow kinds. In particular, the first 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 3.2 shows the graph semantics of a simple configuration (with the black solid arcs) and its information flow diagram (with the gray dashed arcs).

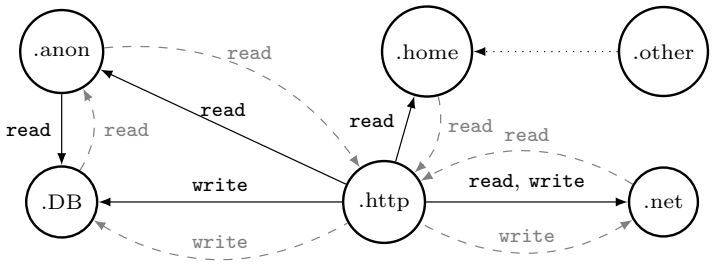


Figure 3.2: A simple configuration (black solid arcs) and its information flow diagram (gray dashed arcs); the dotted arc represents inclusion of the target in the typeattribute of the source.

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 3.3. Intuitively, the entities of type `http` collect information from the network into the database and make data available to the network and to additional entities of type `home`. Information can flow from the network into the database and vice versa, as the configuration satisfies the functional requirements `net +> http +> DB` and `DB +> http +> net` (passing through `anon`). Moreover, the following security requirements are met: `~(DB +> other)` and `DB +> net : DB > anon +> net`. The first states that no information flows from the database to the generic, untrusted types in `other`; the second requirement says that the private information in the database passes through `anon` (where, for example, anonymization takes place) before being delivered in the network.

Formal semantics We formally state 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 if the information flow passes through the specified nodes in the correct order as a result of the designated operations.

Definition 3.1 (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 $\pi \triangleright_I P$, iff

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

$$\pi\,\pi' \triangleright_I P_1\ P_2 \text{ iff } \pi \triangleright_I P_1 \wedge \pi' \triangleright_I P_2$$

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 $\texttt{op} \in o \cap o'$, the arc $(\texttt{n, op, n'})$ can be followed transferring information from $\texttt{n}$ to $\texttt{n'}$. The second case reduces $\texttt{+[}o'\texttt{]>}$ to $\texttt{[}o'\texttt{]>}$. 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, e.g., the first clause can be rewritten as $(\texttt{n}, o, \texttt{n}') \triangleright_I * \texttt{[}o'\texttt{]>}\ \texttt{n}'$ iff $o \cap o' \neq \emptyset$ holds because the kind $*\ \texttt{[}o'\texttt{]>}\ \texttt{n}'$ says that information flows from any node to $\texttt{n}'$.

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

Definition 3.2 (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 be valid w.r.t. $\mathcal{R}$, in symbols $I \models \mathcal{R}$, by cases on the syntax of $\mathcal{R}$ as follows:

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

It is immediate to verify that the requirements on the configuration in Figure 3.2 are indeed satisfied.

Expressivity Path existence constraints express functional requirements, i.e., 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 enforce confidentiality in a Bell-La Padula style, as well as implement the Biba integrity model. Finally, path constraints can express nontransitive properties, like intransitive noninterference. For example $\texttt{n+>n}' : \texttt{n+>n}''\texttt{+>n}'$ requires that the type $\texttt{n}$ cannot transmit any information to $\texttt{n}'$ unless it is done through $\texttt{n}''$.

3.3.2 IFCIL

We introduce the language IFCIL, obtained 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 introduce 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,

```
(macro in_out((type in) (type out))
;IFL; (F1) in +> out ;IFL;
;IFL; (F2) out +> in ;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 3.3: Example of CIL configuration with IFL annotations.

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 make stronger 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 3.3 illustrates both constructs. For example, the second line in Figure 3.3 contains a functional requirement labeled with `(F1)` that requires the existence of a direct or indirect information flow from the node `in` 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 `in`) 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 intuitions apply 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 Section B.3. 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 the preorder $\preceq$, saying that $\mathcal{R}' \preceq \mathcal{R}$ if $\mathcal{R}'$ refines $\mathcal{R}$; the precise definition of $\preceq$ is given in Section B.2.

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

Theorem 3.1 (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

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

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) in +> 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 {`in, out, http, a`} such that I has a single arc (`in, out`), and I' only has the two arcs (`a, http`) and (`http, a`). If they were comparable, Theorem 3.1 would be falsified because $I \models$ `in +> out` but $I \not\models$ `* +> 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` = `in +> 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 `in` 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 3.2, 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 3.3 (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 the 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 3.4 (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 $I \models \mathbb{R}$, iff $I \models \mathcal{R}$ for all $\mathcal{R} \in \mathbb{R}$.

3.4 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 re-use existing verification tools. For this, we first encode a configuration as a Kripke transition system [93] and a requirement as an LTL formula.

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 $L\colon S \rightarrow 2^{AP}$ is a labeling function mapping nodes to a set of propositions that hold at that node. Paths of K are defined as alternating sequences of states and actions starting and ending with a state. In the following, we only consider *maximal* paths, i.e. those paths that cannot be extended.

We associate a IFCL configuration Σ with a KTS with the nodes of Σ as states, 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 3.5 (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., if n is in the typeattribute M

We encode IFL requirements in a suitable version of LTL [93], where the syntax of formulas ϕ is

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

We write $w \models_l \phi$ if the path w of K satifies the LTL formula ϕ; the formal definition can be found in [93]. 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; w satisfies $F(\phi)$ if it

has a state such that the subpath starting from it satisfies ϕ; w satisfies $G(\phi)$ if every subpath starting from a state in w 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 3.3.1 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 3.6 (Encoding of flow kinds)**.** The encoding of flow kinds is defined as follows.

$$\begin{aligned}
&(\!|\mathtt{n\ \ [o]>\ n'}|\!) = \mathtt{n} \wedge \bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}) \wedge X(\mathtt{n}') \\
&(\!|\mathtt{n\ +[o]>\ n'}|\!) = \mathtt{n} \wedge \bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}) \wedge X(U(\bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}), \mathtt{n}')) \\
&(\!|(\mathtt{n\ \ [o]>\ n'})P|\!) = \mathtt{n} \wedge \bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}) \wedge X((\!|P|\!)) \\
&(\!|(\mathtt{n\ +[o]>\ n'})P|\!) = \mathtt{n} \wedge \bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op}) \wedge X(\bigvee_{\mathtt{op} \in \mathtt{o}} (\mathtt{op})\ U\ (\!|P|\!))
\end{aligned}$$

We now lift the satisfiability relation $w \models_l \phi$ from a single path w to the set of *all* the paths of a KTS. This paves the way for defining when a KTS satisfies a set of requirements. To do this, we resort to the predicate $check(K, \phi)$, which given a KTS K and a LTL formula ϕ holds iff $\forall w \in W.w \models_l \phi$, where W is the set of maximal paths in K.

Definition 3.7 (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:

$$\begin{aligned}
K \vdash P &\text{ iff } \neg check(K, \neg(\!|P|\!)) \\
K \vdash \tilde{}P &\text{ iff } check(K, \neg(\!|P|\!)) \\
K \vdash P : P' &\text{ iff } check(K, \neg(\!|P|\!) \vee (\!|P \sqcap P'|\!))
\end{aligned}$$

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

Note that the three clauses above mimic the analogous clauses in the Definition 3.2. The first clause says that no paths satisfy P, which can be expanded as $K \vdash P$ iff $\exists w \in W.w \models_l (\!|P|\!)$, and similarly for the second clause. In the third clause, we consider the meet of P and P', which represents the more general refinement of both P and P'. This is because only maximal paths of K are taken, and if such a path satisfies $P \sqcap P'$ it satisfies both P and P', which is what the requirement asks (when P is satisfied).

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 3.2 (Correctness)**.** 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

$$K \vdash \mathbb{R} \;\Rightarrow\; I \models \mathbb{R}.$$

This theorem enables us to re-use 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 characteristic function for $check(K, \phi)$, introduced above.

As it is known, the worst case complexity of LTL model checking is unfortunately $2^{\mathcal{O}(|\phi|)}\mathcal{O}(|S| + |E'|)$ [93], where $|S|$ and $|E'|$ are the number of nodes and of arcs in the KTS, respectively. In practice we expect the size of the configuration to grow, as well as the number of requirements. However, we do not expect the size of each IFL requirement to grow accordingly.

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

3.4.1 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.

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 deals with CIL's constructs for defining classes and permissions, and reduces the input configuration to one that only uses the fragment

of CIL presented in Section 3.2. Since other constructs, like those about 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 3.3. Its IFCIL semantics is the pair $(G, \mathbb{R} = \{\texttt{F1, F2, F1R, F2R, S1R, S2}\})$, where G is the CIL semantics in Figure 3.2. 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 $\mathcal{R} = \texttt{S1R}$ that require the following $check(I, \neg(\!|\texttt{.DB +> .net}|\!) \vee (\!|\texttt{.DB [read]> .anon +> .net}|\!))$ where:

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

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

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 an extra value `sink` that we will discuss later.
- 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.
- Since NuSMV checks for infinite paths, the special node `sink` is added to N, and an arrow (implicitly labeled with all operations) is added from every node (including `sink`) to it.
- 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 in few seconds.

Validation We experimentally assessed our tool on three real-world CIL policies. The first policy [62] is used in the OpenWrt project, a version of the Linux operating system targeting embedded devices, like network appliances [4]. The second and the third are SELinux example policies, namely *cilbase* [63] and *dspp5* [64], which serve as templates for creating personalized configurations. The analyzed policies have more than ten thousands lines of code, and make an extensive use of all CIL's advanced features, in particular macros and blocks.

Table 3.1: Performance analysis on tree real-world configurations

Requirement kind	Number of requirements	Verification Time
openWRT (45702 lines, 590 types)		
TCB	1	119sec
assured pipeline	3	122sec
wrap untrustworthy	10	100sec
augment only	2	115 sec
total	16	129sec
cilbase (11989 lines, 293 types)		
TCB	1	0.240sec
assured pipeline	4	0.238sec
wrap untrustworthy	6	0.235sec
augment only	2	0.232sec
total	13	0.258sec
dspp5 (14782 lines, 149 types)		
TCB	1	2.22sec
assured pipeline	4	2.14sec
wrap untrustworthy	8	2.28sec
total	13	2.24sec

To assess the expressivity of our language, we encoded in IFL a number of properties that are often considered in the literature, as well as domain-specific policies that we designed.

Our results show that IFCILverif scales well to real world configurations, checking their requirements in a few seconds.

We first consider the following property inspired by Jeager et al. [77], 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 concerns the *assured pipeline* of [65], where the flow from `a` to `z` must pass through a list of intermediate entities `b,c` ... . It suffices to use requirements of the form `a +> z : * +> b +> c +> ... +> *`.

We express the *wrapping of untrustworthy programs* of [120], 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., that `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`.

The results of our analyses on the three configurations are summarized in Table 3.1. For each row the table reports the kind of property, the number of re-

quirements, 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 signals 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 on whether these types are indeed untrusted and on the actual impact the detected violations have on security. This however requires a reverse engineering to better understand the security goals of the analyzed policies.

3.5 Related Work

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, looking for specific 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. [108] propose SELint, a tool for detecting well-known 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. [103] 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 [59]; 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, e.g., 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. [77] 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 our. They neither directly work on structured CIL configurations, nor do 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. [65] 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. Instead, we do not consider exceptions because they can be encoded via typeattributes.

Sarna-Starosta et al. [111] propose a logic-programming based approach to analyze 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. A complete comparison between the two approaches and the efficiency of their implementations requires further investigations.

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. [75] propose Lobster, a high-level DSL for specifying SELinux configurations. This compositional language describes the configuration 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 the configuration. 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 supports 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. [94] propose SEEdit, a security policy configuration sys-

tem that supports creating SELinux configurations using an 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.

3.6 Conclusions and future work

We have proposed IFCIL, a backwards compatible extension of CIL, which has been recently proposed as an intermediate language for SELinux. IFCIL is composed by two ingredients: CIL, for interacting with the system at the low level, precisely enumerating permissions, and IFL, that allow the administrator to express high level information flow requirements, 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, thus granting a binding between the high and low level representations of the system. Our experiments show that the language works well for defining properties that are commonly investigated on SELinux policies, and that the verification times are acceptable even for large real-world configurations.

Discussion We believe that our extension may help with the development of more advanced high-level languages. As our annotations associated with a common intermediate language, they can enrich different high-level languages. Our verification procedure could be used for checking properties when composing code written in different high-level 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 [70]. Moreover, many real CIL configurations only use these 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 discuss in Section 3.4.1, 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 [59].

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. Then, we will provide more friendly diagnostics and suggestions for fixing violated requirements.

We plan to enhance the tool efficiency and response time, and extend 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 [6].

Chapter 4

Collaborative Environments

We now focus on collaborative environments, focusing on granting mutual benefits when evaluating access requests. In a distributed setting each user has a set of his own resources that are possibly shared with others. The access policy protecting these resources is naturally defined by each user in isolation, and independently of the other users. In distributed collaborative contexts, policies should aim at a fair exchange of access grants, so enhancing mutual advantages. For example, different hospitals may share anonymized medical data to improve the quality of statistics. As a further example consider online social networks where Alice decides on her own which users can see her pictures, e.g., only showing them to who share their pictures with her. As a matter of fact, mutuality is the basis of social interactions [52], and it affects how resources are shared.

We address both the case of infinite or reusable assets, where the asset is still available to the owner after he allows access to it, and the case of finite resources that are consumed when exchanged. Examples of finite resources that may be exchanged this way are non fungible tokens, cryptocurrencies, memory storage and computing power, as well as physical assets. Actually, we mainly focus on finite resources, being the most difficult, and recover reusable assets as a special case.

Traditional access control languages do not express conditions that foster mutual benefits, but only check the roles or the attributes of the requester and of the resources. Typically, the exchange of access rights and resources is negotiated by humans and implemented by hand in the access control policy of each contractor. An automatic tool would instead help, which takes access control decisions based on what requesters offer to others. To the best of our knowledge, only few papers have investigated the possibility of expressing a limited form of mutuality in policies, e.g. [118].

Here, we start to address the mutuality issue and propose *MuAC*, an access control system with a logic-based policy language. The MuAC policy language not only allows specifying conditions on resources and users, but also constraints on what requesters are required to deliver in return. Polices of different users affect each other, even though they are defined in isolation and only control

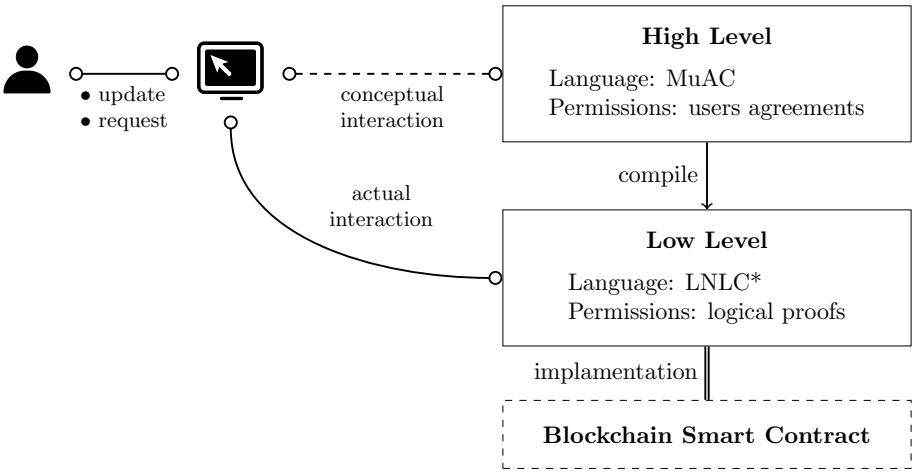


Figure 4.1: Schema of the two-layer approach for MuAC.

the access to the resources of a single user. For example, consider the case of picture sharing mentioned above: Alice needs to be aware of Bob's policy to grant him access to her pictures. In general, deciding an access request may require a complete knowledge of the policies of many or even all the users.

Mutuality may however induce circularity while evaluating access requests. Consider again picture sharing and assume Bob makes a request to view Alice's pictures. If Bob unconditionally allows Alice to watch his, then there is no circularity and Bob's request is accepted. A circularity instead occurs if Bob allows Alice to watch his picture *only if* she shares her pictures with him, as it is the case. Nevertheless, the system should grant Bob access to Alice's pictures, and viceversa.

Unfortunately, classical logic is not fully adequate to express these situations, which are typical of human contracts. To overcome this limitation, a non-standard logic is introduced, and embedded in a mixed logic composed by a linear fragment (for reasoning about resources) and a non-linear fragment (for reasoning about classical access control conditions, like roles and attributes). We therefore exploit this logic to deal with circularity arising in mutual access policies.

In the setting of collaborative environments, we instantiate the two layer approach by proposing a new high level language, the one of MuAC policies, and a logical low level that is more detailed and allows the system to be implemented. The general schema is in Figure 4.1. Summarizing, the two abstraction layers we target are:

- the high level (HL) of MuAC language, speaking of exchanges policies;
- the low level (LL) of a logic-based reasoner implemented as an architecture over blockchain.

The coherence between the high level and low level representations of the system is granted by a compilation from MuAC policies to logical theories. The compilation we propose directly derives from the MuAC logical semantics, and thus it is correct by definition. Since we are not using as low level a preexisting access control technology, we propose an implementation. In accordance with the distributed nature of the context, we opt for an implementation as a blockchain smart contract. Moreover, an off-chain client is given that allows the user to interact with the system at the high level abstracting away low level details, that are entirely in charge of the proposed architecture. The usage of a client also reduces the cost of serving an access request to be linear, which is a necessity given that the user has to pay the execution of his instructions on the blockchain.

Structure of the Chapter

In Section 4.1 we introduce the MuAC system and exemplify the kind of agreements it can mediate; we also give the syntax of the MuAC policy language. In Section 4.2 we present a new linear non-linear contract logic for reasoning about agreements. In Section 4.3 we associate a MuAC configuration (i.e., a mapping between users and their policies) to a logical theory, and formally define the the MuAC system evolution triggered by users requests. In Section 4.4 we show how a MuAC system algorithmically evaluates the requests, and propose an implementation of it on a blockchain. In Section 4.5 we target reusable resources, showing how to adapt the system for this case. In Section 4.6 we compare our proposal with the literature, and concludes in Section 4.7. Additional details and full proofs are in Appendix C.

4.1 Introducing the High Level: MuAC

We introduce the MuAC access control system and its policy language. We intuitively characterize the kind of agreements mediated by MuAC through an example based on blockchain games. We then present the syntax of the policy language.

4.1.1 MuAC Access Control System

The collaborative environment where MuAC comes into play is composed by a set Usr of users, ranged over by $\mathit{usr}, \mathit{usr}', \mathit{usr}''$. Let Res ranged over by $\mathit{res}, \mathit{res}', \mathit{res}''$, be the set of resources in the system; and R, ranged over by r, r', r'', be the set of resource kinds. We assume the function $\mathit{kind} \colon \mathit{Res} \to R$ associates each resource with its kind. Each user owns a non-negative number of resources for each kind. Information about the ownership is stored in the state $\mathfrak{S}$, which, as we will see, changes over time. We assume additional information about the users is stored in the *context* Γ. For example, users may be associated with attributes and may be in relation with other users. We abstractly model

the information stored in Γ as a relation between tuples of users and properties $p \in \mathbb{P}$ that we leave unspecified. Each user $usr \in Usr$ defines a *MuAC policy* Σ_{usr}, using the MuAC policy language defined later. Let Pol be the set of all MuAC policies, then a *MuAC configuration* Σ associates users with their policies $\Sigma_{usr} \in Pol$. Formally:

Definition 4.1 (MuAC system)**.** A MuAC system over Usr and Res is a pair (Γ, Σ), with

$$\Gamma \subseteq \bigcup_{n \in \mathbb{N}} Usr^n \times \mathbb{P}$$
$$\Sigma \colon Usr \to Pol$$

A MuAC state is a function $\mathfrak{S} \colon Usr \to 2^{Res}$

As usual for access control systems, the user usr can make a request for a resource of kind r. To decide if the request is to be granted or denied, the system first check if usr owns a resource r. When this is not the case, MuAC policies are considered. In their MuAC policies, users specify in isolation what they want in return for granting other users access to their own resources. If an agreement is found, the request is served, i.e., the MuAC state changes from $\mathfrak{S}$ to a new state $\mathfrak{S}'$ where a resource of kind r is associated with usr. Formally:

Definition 4.2 (MuAC system evolution)**.** Let $\mathcal{S}$ be the set of states $\mathfrak{S}$. Given a MuAC system (Γ, Σ), its evolution is a relation

$$\rightsquigarrow_{\Gamma,\Sigma} \subseteq \mathcal{S} \times Usr \times R \times \mathcal{S}$$

such that if $(\mathfrak{S}, usr, r, \mathfrak{S}') \in \rightsquigarrow_{\Gamma,\Sigma}$ then $res \in Res$ exists such that $kind(res) = r$ and $res \in \mathfrak{S}'(usr)$.

We call *MuAC request* a pair (usr, r), and we represent it as $r@usr?$. Moreover, we write

$$\Gamma, \Sigma \vDash \mathfrak{S} \rightsquigarrow^{r@usr?} \mathfrak{S}'$$

for $(\mathfrak{S}, usr, r, \mathfrak{S}') \in \rightsquigarrow_{\Gamma,\Sigma}$.

As stated before, users do not have to check policies and bargain on their own, their policies are automatically checked to find agreements that are mutually advantageous. This also means that they are not required being online while the process takes place.

We exemplify the intended usage of MuAC in a simple scenario.

4.1.2 Running Example

A blockchain game is a video game where the ownership of virtual objects is proved by the association of a *non fungible token* (NFT) with the account of the players. Usually, in online games players can exchange their objects each other, with other virtual objects or the on-game currency. In a blockchain game,

the exchanges are not implemented by the programmers of the game. The players can sell or exchange them on their own directly using the blockchain in a transparent manner. For convenience, we only focus on exchanges of virtual objects.

We consider a fictional game, played by Alice, Bob and Carl. As it is common in online games, players can join guilds of adventurers; Bob and Carl belongs to the guild called *paladins*. The resource that players can trade are on-game items found while playing: healing potions (*hp* in the following), spell books (*sb* in the following), light and heavy weapons (*lw* and *hw* in the following). Summing up, $Usr = \{Alice, Bob, Carl\}$, $R = \{hp, sb, lw, hw\}$, and Γ stores that Bob and $Carl$ are in the *paladin* guild.

The rules in the MuAC configurations of Alice (rules **A1**, **A2**), Bob (rules **B1**, **B2**, **B3**) and Carl (rules **C1**, **C2**, **C3**) state the following:

A1 I give *sb* if I get *hw* in return;

A2 I give *sb* if I get *hp* in return;

B1 I give *lw* if I get *sb* in return;

B2 I give you a *hp* if you give me a *sb*;

B3 If you are a paladin, I give you a *lw* if you give me a *hp*;

C1 I give *hw* if I get *lw* in return;

C2 I give you a *hp* if you give me a *lw*;

C3 If you give a *sb* to a paladin, I give you an *hp*.

Finally, assume that the current state $\mathfrak{S}$ is such that Alice has one *sb*, Bob has one *lw*, and Carl has both one *hw* and one *hp*.

We present a series of request for showing the kind of agreements that may arise in MuAC, in increasing complexity.

Example 4.1 (Direct exchange)**.** The simplest case is when the resources of two players are exchanged. If Bob makes a request for *hp*, the system realizes that he has none of them and start considering the MuAC configuration, looking for exchanges. Carl is willing to exchange *hp* with a *lw* (rule **C2**). Bob has a *lw*, and he is willing to exchange it with a *hp*, but only with a paladin (rule **B3**). Carl is a paladin, thus the agreement is met, and the exchange takes place. Bob will have the *hp* he needs, and Carl will have a *lw*. Note that the agreement is satisfactory for both parts, and no bargaining is needed.

Example 4.2 (I pay for you)**.** In this case, the rule **C3** comes into play. Note that Carl accept to pay for other members of the paladin gild. If Bob makes a request for *sb*, the system start considering the MuAC configuration. Alice offers a *sb* in return for *hp* (rule **A2**), regardless of who has to give the *hp*. Bob has no *hp* to exchange for *sb* with Alice, but luckily he is in the paladin guild, thus Carl is willing to pay (rule **C3**). Bob takes the *sb* of Alice, and Alice the *hp* of Carl.

Example 4.3 (Circular Exchange)**.** Assume Alice requests *hw*. She offers *sb* in return (rule **A1**), but none is willing to make such an exchange. The only one that offers *hw* is Carl, who wants *lw* in return (rule **C1**). Alice has no rule for giving *lw*, and no *lw* resource. No agreement is possible between any two users, but if Bob comes into play then an exchange is possible. Bob sells *lw* for *sb* (rule **B1**). Everyone is happy if Alice gives her *sb* to Bob (satisfying the condition of rule **B1**), Bob gives his *lw* to Carl (satisfying the condition of rule **C1**), and Carl gives his *hw* to Alice (satisfying the condition of rule **A1**). In practice, every user *usr* is paying for some other user usr', provided that some usr'' is paying for *usr*. It is trivial to verify that everyone is happy: they are receiving what they wanted by paying what they promised.

Example 4.4 (Resource Supplier)**.** Our last case is an agreement between two parts that would be reachable, but one of the two has not the needed resource. Assume Alice wants *hp*. A simple agreement would be between her and Bob, Alice gives *sb* for *hp* (rule **A2**), and Bob gives *hp* for *sb* (rule **B2**). Unfortunately, Bob has no *hp*, but in spite of that the system finds an agreement where Alice gets the *hp* and Bob the *sb*. Indeed, Bob has *lw* that can be exchanged with Carl for *hp* (rule **C2**), and Bob agrees since Carl is in his guild (rule **B3**). Thus Carl take *lw* from Bob and gives him *hp*, that Carl exchange with Alice for *sb*.

4.1.3 MuAC Syntax

Let U, ranged over by u, u', u'', be the set of user variables with two distinguished elements `Me` and `Requester`, where `Me` represents the owner of the policy and `Requester` represents the user issuing an access request. Recall that p, p', p'' are properties in $\mathbb{P}$.

Every user $usr \in Usr$ defines in isolation his policy Σ_{usr} as a set of rules $\nu \in V$ given by the following grammar:

$$\begin{aligned} \nu &::= r\colon P\ \mathit{GiveLs} \\ P &::= \mathtt{p}(u, \ldots, u)\ P \mid \epsilon \\ \mathit{GiveLs} &::= \mathtt{Gives}(u, r, u')\ \mathit{GiveLs} \mid \epsilon \end{aligned}$$

Intuitively, a rule associates a resource kind r with a (possibly empty) list of properties and a list of *exchanges* asked in return, where `Gives(u, r, u')` states that u' is required to give to u a resource r.

Example 4.5. We know express in MuAC the rules of Alice, Bob and Carl in the running example of subsection 4.1.2. The rules of Alice are:

```
spell_book : Gives(Me, heavy_weapon, u)                    // Rule A1
spell_book : Gives(Me, healing_potion, u)                  // Rule A2
```

The rules of Bob are:

```
light_weapon : Gives(Me, spell_book, u)                    // Rule B1
healing_potion : Gives(Me, spell_book, Requester)          // Rule B2
light_weapon : is_paladin(Requester),                      // Rule B3
                Gives(Me, healing_potion, Requester)
```

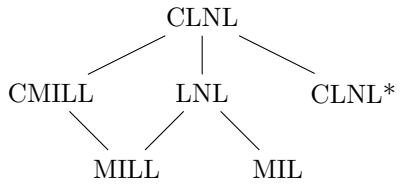


Figure 4.2: Inclusion between logic systems, growing upwards. MILL and MIL are the multiplicative fragments of linear and intuitionistic Logic, respectively; LNL is linear non-linear logic; and CLNL* is the computational fragment of our multiplicative, linear non-linear, contractual logic CLNL.

Finally, the ones of Carl follows

```
heavy_weapon : Gives(Me, light_weapon, u)                    // Rule C1
healing_potion : Gives(Me, light_weapon, Requester)          // Rule C2
healing_potion : is_paladin(u),                              // Rule C3
                 Gives(u, spell_book, Requester)
```

(the text after `//` is a comment.)

4.2 The Low Level: A Logic for MuAC

We give here a logic for characterizing the agreements and exchanges of MuAC systems. To do that, we use two auxiliary ingredients. The first one is a new operator that enables expressing promises in linear logic for dealing with contractual behaviour. More precisely, we take *MILL*, the multiplicative fragment of intuitionistic linear logic [58] (the one that only deals with $\otimes$ and $\multimap$). Then we extend it with a new operator for modeling promises, i.e. guarantees that something will hold provided that the given conditions are satisfied, obtaining CMILL.

The second ingredient deals with non-linear conditions that the user can express in MuAC policies (like the membership in a guild in our running example of subsection 4.1.2). For doing so, we combine CMILL and non-linear logic, following the line of linear non-linear logic, *LNL* [25], obtaining CLNL.

The whole CLNL is not needed for characterizing MuAC systems, thus we present a computational fragment, called CLNL* that is adequate and has good computational properties. The inclusion between the logic systems is schematized in Figure 4.2.

4.2.1 Contractual Linear Implication

Here we define Contractual Multiplicative Linear Logic, CMILL for short, an extension of MILL with the new operator $\multimap\!\!\!\circ$, called *linear contractual implication*, or *contract* for short. We have been inspired by Bartoletti and Zunino's PCL [20]. Roughly, a formula $\varphi \multimap\!\!\!\circ \varphi'$ states the promise that φ' will eventually

$$\frac{}{A \vdash A}\,(\text{Ax}) \qquad \frac{\Phi \vdash \varphi}{\Phi, I \vdash \varphi}\,(I\text{-left}) \qquad \frac{}{\vdash I}\,(I\text{-right})$$

$$\frac{\Phi, \varphi, \varphi' \vdash \varphi''}{\Phi, \varphi \otimes \varphi' \vdash \varphi''}\,(\otimes\text{-left}) \qquad \frac{\Phi \vdash \varphi \quad \Phi' \vdash \varphi'}{\Phi, \Phi' \vdash \varphi \otimes \varphi'}\,(\otimes\text{-right})$$

$$\frac{\Phi \vdash \varphi \quad \Phi', \varphi' \vdash \varphi''}{\Phi, \Phi', \varphi \multimap \varphi' \vdash \varphi''}\,(\multimap\text{-left}) \qquad \frac{\Phi, \varphi \vdash \varphi'}{\Phi \vdash \varphi \multimap \varphi'}\,(\multimap\text{-right})$$

$$\frac{\Phi \vdash \varphi \quad \Phi', \varphi \vdash \varphi'}{\Phi, \Phi' \vdash \varphi'}\,(\text{Cut})$$

$$\frac{\Phi \vdash \varphi'}{\Phi \vdash \varphi \multimap\!\!\!\circ\, \varphi'}\,(\multimap\!\!\!\circ\text{-zero}) \qquad \frac{\varphi'' \vdash \varphi \otimes \varphi''' \quad \Phi, \varphi' \vdash \varphi''}{\Phi, \varphi \multimap\!\!\!\circ\, \varphi' \vdash \varphi''}\,(\multimap\!\!\!\circ\text{-fix})$$

Figure 4.3: CMILL rules.

hold provided that φ is *true*. The syntax of CMILL is as follows, where I stands for *true* and A is an atomic proposition:

$$\varphi ::= I \mid A \mid \varphi \otimes \varphi \mid \varphi \multimap \varphi \mid \varphi \multimap\!\!\!\circ\, \varphi.$$

The operators $\otimes$ and $\multimap$ are the multiplicative conjunction and implication of linear logic. Roughly, $\varphi \otimes \varphi'$ holds if and only if both φ and φ' hold, and $\varphi \multimap \varphi'$ says that we can derive φ' if φ holds, but after the derivation both the implication and φ are no longer available. A common way of describing the meaning of linear logic is to consider formulas as resources that can be used only once: $\varphi \otimes \varphi'$ means that we have both resources, and $\varphi \multimap \varphi'$ means that we can consume φ and the implication to obtain φ'.

The inference rules are those of MILL, plus the ones for $\multimap\!\!\!\circ$, see Figure 4.3.

In [20], Bartoletti and Zunino suggest 11 properties that a sensible contractual implication must satisfy to accurately model contractual reasoning. We briefly discuss these properties below, grouping them in three classes: *handshaking*, *standard implication*, *weakening and strengthening*. Seven of these properties straightforwardly hold for CMILL and additional three do after some slight adjustments to CMILL. We argue that the remaining two properties are not desirable in the context of linear logic.

Handshaking

Four handshaking properties hold for CMILL without any adjustment. The *simple handshaking property* intuitively states that if two contracting parties have a mutual agreement, then what they promise must hold:

$$\vdash (\varphi \multimap\!\!\!\circ\, \varphi') \otimes (\varphi' \multimap\!\!\!\circ\, \varphi) \multimap \varphi \otimes \varphi' \tag{4.1}$$

The *circular handshaking property* generalizes property (4.1) to a case in which n parts circularly rely one on the promises of another:

$$\vdash (\varphi_1 \multimap\!\!\!\circ\, \varphi_2) \otimes ... \otimes (\varphi_{n-1} \multimap\!\!\!\circ\, \varphi_n) \otimes (\varphi_n \multimap\!\!\!\circ\, \varphi_1) \multimap \varphi_1 \otimes ... \otimes \varphi_n \tag{4.2}$$

As a corollary, also the *one-step circular handshake* works:

$$\vdash (\varphi \multimap\!\!\!\circ\, \varphi) \multimap \varphi \tag{4.3}$$

Finally, in the *greedy handshaking*, each party i promises φ_i only provided that all the other parties promise φ_j with $j \neq i$:

$$\vdash \bigotimes_{i \in 1...n} (\bigotimes_{j \neq i} \varphi_j \multimap\!\!\!\circ\, \varphi_i) \multimap \varphi_1 \otimes \cdots \otimes \varphi_n \tag{4.4}$$

Of course, the condition of a promise $\varphi \multimap\!\!\!\circ\, \varphi'$ can also be directly satisfied, i.e. when φ holds. This property is expressed in [20] as

$$\not\vdash \varphi \otimes (\varphi \multimap\!\!\!\circ\, \varphi') \multimap \varphi'$$

In CMILL the validity of this formula is not desirable, because it discards φ in contrast with the intuition behind linear logic. To keep linearity, we then adjust the above and obtain:

$$\vdash \varphi \otimes (\varphi \multimap\!\!\!\circ\, \varphi') \multimap \varphi \otimes \varphi' \tag{4.5}$$

As a consequence, the linear contractual implication is not stronger than (i.e., it does not imply) linear implication, just as the contractual implication proposed by [20] is stronger than standard implication, in symbols:

$$\not\vdash (\varphi \multimap\!\!\!\circ\, \varphi') \multimap (\varphi \multimap \varphi')$$

To take care that intuitively φ cannot be discarded from the conclusions of the contractual implication, we propose the following similar property:

$$\vdash (\varphi \multimap\!\!\!\circ\, \varphi') \multimap (\varphi \multimap \varphi \otimes \varphi') \tag{4.6}$$

Standard implication properties

Two properties of the standard implication listed in [20] also hold for linear contractual implication, while a third one requires a minor adjustment.

First, also in CMILL a contract that promises *true* (I) is always satisfied.

$$\vdash \varphi \multimap\!\!\!\circ\, I \tag{4.7}$$

Transitivity does not hold as it is for linear contractual implication:

$$\not\vdash (\varphi \multimap\!\!\!\circ\, \varphi') \otimes (\varphi' \multimap\!\!\!\circ\, \varphi'') \multimap (\varphi \multimap\!\!\!\circ\, \varphi'')$$

and we think that this property is not desirable in a *linear* contractual logic. As a matter of fact, assume transitivity, take $\varphi'' = \varphi$ in the formula above and apply the property (4.3). You get the following formula that contradicts the basic assumption of linear logic, because an occurrence of φ' is discarded in the right-hand side of $\multimap$:

$$(\varphi \multimap\!\!\!\circ\, \varphi') \otimes (\varphi' \multimap\!\!\!\circ\, \varphi) \multimap \varphi$$

The following example shows what can go wrong if transitivity of $\multimap\!\!\!\circ$ is assumed.

Example 4.6. Consider the following formula φ that involves three users, Alice, Bob and Carl that interact for sharing resources.

$$\varphi := (A_1 \multimap\!\!\!\circ\, A_2) \otimes (A_2 \multimap\!\!\!\circ\, A_3) \otimes (A_3 \multimap\!\!\!\circ\, A_1)$$

And let the proposition A_1 mean that Alice gives a spell book *sb* to Bob; let A_2 mean that Bob gives a light weapon *lw* to Carl; let A_3 mean that Carl gives a heavy weapon *hw* to Alice. Clearly, each user promises to give a resource if they get another in return. One can apply the circular handshake property (4.2) and derive $\varphi \multimap A_1 \otimes A_2 \otimes A_3$, i.e., all the exchanges happen, obtaining a fair treatment of all the users. However, if we assume transitivity of $\multimap\!\!\!\circ$, one could also derive $\varphi \multimap A_1 \otimes A_3$, where the treatment of Carl and Bob is unfair (Bob gets a free resource, and Carl gets nothing for his *hw*).

Nevertheless, we recover a similar property in the following adjusted form, in which the "intermediate" formula φ' is kept:

$$\vdash (\varphi \multimap\!\!\!\circ\, \varphi') \otimes (\varphi' \multimap\!\!\!\circ\, \varphi'') \multimap (\varphi \multimap\!\!\!\circ\, \varphi' \otimes \varphi'') \tag{4.8}$$

The third property holds as it is in PCL, and states that if a promise φ' is *true*, then also any contract that promises φ' is *true*:

$$\vdash \varphi' \multimap (\varphi \multimap\!\!\!\circ\, \varphi') \tag{4.9}$$

Weakening and strengthening

In PCL, the object of a promise can be arbitrarily weakened, and the pre-condition can be arbitrarily strengthened, but we think such an unconstrained loosening is not adequate in a linear contractual logic, i.e.

$$\nvdash (\varphi' \multimap \varphi) \otimes (\varphi \multimap\!\!\!\circ\, \varphi'') \multimap (\varphi' \multimap\!\!\!\circ\, \varphi'')$$
$$\nvdash (\varphi \multimap\!\!\!\circ\, \varphi'') \otimes (\varphi'' \multimap \varphi''') \multimap (\varphi \multimap\!\!\!\circ\, \varphi''')$$

Indeed, would they have held, also the following would:

$$(\varphi \multimap\!\!\!\circ\, \varphi') \otimes (\varphi \otimes \varphi' \multimap\!\!\!\circ\, \varphi'') \multimap (\varphi \multimap\!\!\!\circ\, \varphi'')$$

with the effect of discarding φ' from the agreement, and of possibly distorting the meaning of the involved promises, as shown by the example below.

Example 4.7. Consider now the following theory Φ':

$$A_1 \multimap A_2, A_1 \otimes A_2 \multimap A_3, A_3 \multimap A_1$$

One can apply the greedy circular handshake property (4.2), obtaining $\Phi \vdash A_1 \otimes A_2 \otimes A_3$, which treats each user in a balanced way. However, if we assume weakening and strengthening to hold, one could also derive $\Phi' \vdash A_1 \otimes A_3$, where the treatment of Carl and Bob is unfair (Bob gets a free resource, and Carl gets nothing for his *hw*).

Weakening the object of a promise and strengthening its preconditions are however two properties that may be interesting to have in a disciplined form in a linear contractual logic. A sensible formulation requires a deep investigation and we leave it as future work.

4.2.2 Contractual Linear Non-Linear Logic

Following the approach of [25], we combine CMILL and MIL, obtaining CLNL. The formulas φ of CLNL obey the followig grammar:

$$\varphi ::= I \mid A \mid \varphi \otimes \varphi \mid \varphi \multimap \varphi \mid \varphi \multimap\!\!\!\!\circ\ \varphi \mid F\psi$$
$$\psi ::= \top \mid X \mid \psi \wedge \psi \mid \psi \to \psi \mid G\varphi$$

Intuitively, I and $\top$ stand for linear and non-linear *true*; A and X are linear and non-linear atomic propositions; and, as in LNL, the operator F "lifts" a non-linear formula ψ to a linear one φ (this is the only difference with CMILL formulas) while G does the opposite.

The CLNL inference rules are in Figure 4.4. They include those of LNL, as presented in [25], plus the ones for the contractual implication.

As for LNL, also CLNL judgments are of two kinds:

$$\Psi \Vdash \psi \qquad\qquad \Psi; \Phi \vdash \varphi$$

Roughly, the left one is for non-linear and the right one for linear reasoning. In the judgments, Ψ and Φ are multisets of non-linear and linear formulae, respectively. We feel free to omit an empty multiset.

Computational Fragment

We present the computational fragment of CLNL, CLNL*, restricting sequents to the specific form introduced below. The logic CLNL* supports the evaluation of the access control language we are proposing, and for this reason we call it *computational*. Its elements are defined follow.

Let Ω, Θ, Δ and $\mathbb{S}$, be multisets of CLNL propositions ω, θ, δ and S respectively, defined as

$$\omega ::= \top \mid X \mid \omega \wedge \omega \mid \omega \to \omega \mid G\theta \mid G\delta$$
$$\theta ::= \delta \multimap\!\!\!\!\circ\ \delta$$
$$\delta ::= I \mid A \multimap A \mid \delta \otimes \delta$$
$$S ::= I \mid A \mid S \otimes S$$

$$\frac{}{\Vdash \top}\,(\text{C-}\top) \qquad \frac{}{\psi \Vdash \psi}\,(\text{C-Ax}) \qquad \frac{}{\Psi; \varphi \vdash \varphi}\,(\text{L-Ax})$$

$$\frac{\Psi, \psi, \psi \Vdash \psi'}{\Psi, \psi \Vdash \psi'}\,(\text{C-Cont}) \qquad \frac{\Psi, \psi, \psi; \Phi \vdash \varphi}{\Psi, \psi; \Phi \vdash \varphi}\,(\text{L-Cont})$$

$$\frac{\Psi \Vdash \psi'}{\Psi, \psi \Vdash \psi'}\,(\text{C-Weak}) \qquad \frac{\Psi; \Phi \vdash \varphi}{\Psi, \psi; \Phi \vdash \varphi}\,(\text{L-Weak}) \qquad \frac{\Psi \Vdash \psi \quad \Psi' \Vdash \psi'}{\Psi, \Psi' \Vdash \psi \wedge \psi'}\,(\wedge\text{-right})$$

$$\frac{\Psi, \psi \Vdash \psi''}{\Psi, \psi \wedge \psi' \Vdash \psi''}\,(\text{C-}\wedge\text{-left1}) \qquad \frac{\Psi, \psi' \Vdash \psi''}{\Psi, \psi \wedge \psi' \Vdash \psi''}\,(\text{C-}\wedge\text{-left2})$$

$$\frac{\Psi, \psi; \Phi \vdash \varphi}{\Psi, \psi \wedge \psi'; \Phi \vdash \varphi}\,(\text{L-}\wedge\text{-left1}) \qquad \frac{\Psi, \psi'; \Phi \vdash \varphi}{\Psi, \psi \wedge \psi'; \Phi \vdash \varphi}\,(\text{L-}\wedge\text{-left2})$$

$$\frac{\Psi \Vdash \psi \quad \Psi', \psi' \Vdash \psi''}{\Psi, \psi \to \psi', \Psi' \Vdash \psi''}\,(\text{C-}\to\text{-left}) \qquad \frac{\Psi, \psi \Vdash \psi'}{\Psi \Vdash \psi \to \psi'}\,(\text{C-}\to\text{-right})$$

$$\frac{\Psi \Vdash \psi \quad \Psi', \psi'; \Phi \vdash \varphi}{\Psi, \psi \to \psi', \Psi'; \Phi \vdash \varphi}\,(\text{L-}\to\text{-left})$$

$$\frac{\Psi \Vdash \psi \quad \Psi', \psi \Vdash \psi'}{\Psi, \Psi' \Vdash \psi'}\,(\text{CC-cut}) \qquad \frac{\Psi \Vdash \psi \quad \Psi', \psi; \Phi \vdash \varphi}{\Psi, \Psi'; \Phi \vdash \varphi}\,(\text{CL-cut})$$

$$\frac{\Psi; \Phi \vdash \varphi \quad \Psi'; \Phi', \varphi \vdash \varphi'}{\Psi, \Psi'; \Phi, \Phi' \vdash \varphi'}\,(\text{LL-cut})$$

$$\frac{\Psi; \Phi, \varphi \vdash \varphi'}{\Psi, G\varphi; \Phi \vdash \varphi'}\,(\text{G-left}) \qquad \frac{\Psi \vdash \varphi}{\Psi \Vdash G\varphi}\,(\text{G-right})$$

$$\frac{\Psi, \psi; \Phi \vdash \varphi}{\Psi; \Phi, F\psi \vdash \varphi}\,(\text{F-left}) \qquad \frac{\Psi \Vdash \psi}{\Psi \vdash F\psi}\,(\text{F-right})$$

$$\frac{\Psi; \Phi \vdash \varphi}{\Psi; \Phi, I \vdash \varphi}\,(I\text{-left}) \qquad \frac{}{\vdash I}\,(I\text{-right})$$

$$\frac{\Psi; \Phi \vdash \varphi \quad \Psi; \Phi', \varphi' \vdash \varphi''}{\Psi; \Phi, \Phi', \varphi \multimap \varphi' \vdash \varphi''}\,(\multimap\text{-left}) \qquad \frac{\Psi; \Phi, \varphi \vdash \varphi'}{\Psi; \Phi \vdash \varphi \multimap \varphi'}\,(\multimap\text{-right})$$

$$\frac{\Psi; \Phi, \varphi, \varphi' \vdash \varphi''}{\Psi; \Phi, \varphi \otimes \varphi' \vdash \varphi''}\,(\otimes\text{-left}) \qquad \frac{\Psi; \Phi \vdash \varphi \quad \Psi; \Phi' \vdash \varphi'}{\Psi; \Phi, \Phi' \vdash \varphi \otimes \varphi'}\,(\otimes\text{-right})$$

$$\frac{\Psi; \Phi \vdash \varphi'}{\Psi; \Phi \vdash \varphi \multimap \varphi'}\,(\multimap\text{-zero}) \qquad \frac{\Psi; \varphi'' \vdash \varphi \otimes \varphi''' \quad \Psi; \Phi, \varphi' \vdash \varphi''}{\Psi; \Phi, \varphi \multimap \varphi' \vdash \varphi''}\,(\multimap\text{-fix})$$

Figure 4.4: CLNL rules.

From the computational point of view, an element of Ω represents non-linear knowledge; an element of Θ a contract; an element of Δ an agreed exchange; finally, S and the elements of $\mathbb{S}$ are linear conjunctions of atomic predicates, representing the next and actual states of the computation.

Definition 4.3. A LNCL sequent is *computational* if it is of the following form

$$\Omega; \Theta, \Delta, \mathbb{S} \vdash S.$$

where Ω is the non-linear part Ψ; Φ is split into Θ, Δ and $\mathbb{S}$; and the succedent φ is S.

A computational sequent is *initial* if $\Theta, \Delta = \emptyset$, i.e., if it is $\Omega; \mathbb{S} \vdash S$.

The rules of CLNL* are in Figure 4.5. We abuse notation: tensor products are seen as multisets, given that $\otimes$ is associative and commutative; and the symbol $\subseteq$ acts also as multiset inclusion. Most of the inference rules are inherited from the ones of CLNL but with constraints on the form of the used predicates, that make some rules useless. Linear right rules are not considered, apart from the one of $\otimes$. This is because in computational sequents only linear conjunctions of atomic propositions are legal to the right of $\vdash$. Of course, no F rule is needed since this operator is absent in CLNL*. Finally, the rules for $\multimap$ are specific of this fragment, and we briefly discuss them below. Rule ($\multimap$-Spend) is a weakened version of ($\multimap$-Fix) that fits the syntactic constraints on CLNL* formulas. Rule ($\multimap$-Merge) merges the left and right parts of two propositions in Θ. Maybe surprisingly, note that ($\multimap$-Merge) is not valid in CLNL, but if used in CLNL* produces only CLNL theorems, due to the additional constraints on the syntax, as proved by the following theorem.

The inference rules of CLNL* are correct with respect to computational sequents of CLNL, CLNL* thus inherits the correctness properties from CLNL. Formally:

Theorem 4.1 (CLNL* immersion)**.** For all Ω, Θ, Δ, $\mathbb{S}$ and S,

$$\Omega; \Theta, \Delta, \mathbb{S} \vdash S$$

is valid in CLNL* only if it is valid in CLNL.

A necessary condition for CLNL* to be adequate as semantics for MuAC is that of decidability, which is stated for initial sequents in the following.

4.2.3 Deciding CLNL*

We target the problem of deciding whether an initial computational sequent is valid or not. First we define two normal forms for proofs, and show that they are general, i.e., a proof exists in CLNL* for an initial sequent only if a proof in normal form exists. Then we reduce the problem of finding a proof in the first normal form to reachability in Petri Nets, which is known to be decidable. Finally, we reduce the problem of finding a proof in the second normal form to a proof in the first normal form.

$$\frac{}{\Vdash \top}\,(\text{C-}\top) \qquad \frac{}{\omega \Vdash \omega}\,(\text{C-Ax}) \qquad \frac{}{\Omega; A \vdash A}\,(\text{L-Ax})$$

$$\frac{\Omega, \omega, \omega \Vdash \omega'}{\Omega, \omega \Vdash \omega'}\,(\text{C-Cont}) \qquad \frac{\Omega, \omega, \omega; \Theta, \Delta, \mathbb{S} \vdash S}{\Omega, \omega; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{L-Cont})$$

$$\frac{\Omega \Vdash \omega'}{\Omega, \omega \Vdash \omega'}\,(\text{C-Weak}) \qquad \frac{\Omega; \Theta, \Delta, \mathbb{S} \vdash S}{\Omega, \omega; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{L-Weak})$$

$$\frac{\Omega, \omega \Vdash \omega''}{\Omega, \omega \wedge \omega' \Vdash \omega''}\,(\text{C-}\wedge\text{-left1}) \qquad \frac{\Omega, \omega' \Vdash \omega''}{\Omega, \omega \wedge \omega' \Vdash \omega''}\,(\text{C-}\wedge\text{-left2}) \qquad \frac{\Omega \Vdash \omega \quad \Omega' \Vdash \omega'}{\Omega, \Omega' \Vdash \omega \wedge \omega'}\,(\text{C-}\wedge\text{-right})$$

$$\frac{\Omega, \omega; \Theta, \Delta, \mathbb{S} \vdash S}{\Omega, \omega \wedge \omega'; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{L-}\wedge\text{-left1}) \qquad \frac{\Omega, \omega'; \Theta, \Delta, \mathbb{S} \vdash S}{\Omega, \omega \wedge \omega'; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{L-}\wedge\text{-left2})$$

$$\frac{\Omega \Vdash \omega \quad \Omega', \omega' \Vdash \omega''}{\Omega, \omega \to \omega', \Omega' \Vdash \omega''}\,(\text{C-}\to\text{-left}) \qquad \frac{\Omega, \omega \Vdash \omega'}{\Omega \Vdash \omega \to \omega'}\,(\text{C-}\to\text{-right})$$

$$\frac{\Omega \Vdash \omega \quad \Omega', \omega'; \Theta, \Delta, \mathbb{S} \vdash S}{\Omega, \omega \to \omega', \Omega'; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{L-}\to\text{-left}) \qquad \frac{\Omega \Vdash \omega \quad \Omega', \psi; \Theta, \Delta, \mathbb{S} \vdash S}{\Omega, \Omega'; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{CL-cut})$$

$$\frac{}{\vdash I}\,(I\text{-right}) \qquad \frac{\Omega; \Theta, \Delta, \mathbb{S} \vdash S \qquad \Omega; \Theta', \Delta', \mathbb{S}', S' \vdash S''}{\Omega; \Theta, \Theta', \Delta, \Delta', \mathbb{S}, \mathbb{S}', S \multimap S' \vdash S''}\,(\multimap\text{-left})$$

$$\frac{\Omega; \Theta, \Delta, \mathbb{S} \vdash S \qquad \Omega; \Theta', \Delta', \mathbb{S}' \vdash S'}{\Omega; \Theta, \Theta', \Delta, \Delta', \mathbb{S}, \mathbb{S}' \vdash S \otimes S'}\,(\otimes\text{-right}) \qquad \frac{\Omega; \Theta, \theta, \theta', \Delta, \mathbb{S} \vdash S}{\Omega; \Theta, \theta \otimes \theta', \Delta, \mathbb{S} \vdash S}\,(\otimes\text{-left-}\Theta)$$

$$\frac{\Omega; \Theta, \Delta, \delta, \delta', \mathbb{S} \vdash S}{\Omega; \Theta, \Delta, \delta \otimes \delta', \mathbb{S} \vdash S}\,(\otimes\text{-left-}\Delta) \qquad \frac{\Omega; \Theta, \Delta, \mathbb{S}, S', S'' \vdash S}{\Omega; \Theta, \Delta, \mathbb{S}, S' \otimes S'' \vdash S}\,(\otimes\text{-left-}\mathbb{S})$$

$$\frac{\Omega; \Theta, \theta, \Delta, \mathbb{S} \vdash S}{\Omega, G\theta; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{G-left-}\theta) \qquad \frac{\Omega; \Theta, \Delta, \delta, \mathbb{S} \vdash S}{\Omega, G\delta; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{G-left-}\delta)$$

$$\frac{\delta \subseteq \delta' \quad \Omega; \Theta, \Delta, \delta', \mathbb{S} \vdash S}{\Omega; \Theta, \delta \multimap \delta', \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Spend}) \qquad \frac{\Omega; \Theta, \delta \otimes \delta'' \multimap \delta' \otimes \delta''', \Delta, \mathbb{S} \vdash S}{\Omega; \Theta, \delta \multimap \delta', \delta'' \multimap \delta''', \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Merge})$$

Figure 4.5: CLNL* rules.

Let Sr, Cr, Lr, Gr, Pr be set of CLNL* rules defined as follows.

$$
\begin{aligned}
Sr &= \{\text{(L-Weak), (L-Cont)}\} \\
Cr &= \{\text{(C-}\top\text{), (C-Ax), (C-Cont), (C-Weak), (C-}\wedge\text{-left1), (C-}\wedge\text{-left2),} \\
&\quad \text{(C-}\to\text{-left), (C-}\to\text{-right), (L-}\wedge\text{-right),(L-}\wedge\text{-left2),(L-}\to\text{-left), (CL-Cut)}\} \\
Lr &= \{(\multimap\text{-left}),(\otimes\text{-right}),(\otimes\text{-left-}\Theta),(\otimes\text{-left-}\Delta),(\otimes\text{-left-}\mathbb{S})\} \\
Gr &= \{\text{(G-left-}\theta\text{), (G-left-}\delta\text{)}\} \\
Pr &= \{(\multimap\text{-Spend}), (\multimap\text{-Merge})\}
\end{aligned}
$$

In the following, we will call *proof* the derivation of a theorem from the axioms, and only use the term *derivation* for a derivation with open assumptions, i.e., a proof tree where the leaves are not only axioms. Moreover, let A be a set of rules, we then write Π_A for a CLNL* proof or derivation that only applies rules in A. We also write Ω_G for a multiset only containing linear proposition θ or δ preceded by G.

Definition 4.4 (CLNL* normal forms)**.** A proof Π for a sequent $\Omega; \mathbb{S} \vdash S$ in CLNL* is *normal* if it can be decomposed in either form

$$
\dfrac{\Pi_{Lr\cup\{\text{(L-Ax), (I-right)}\}}}{\begin{array}{c}\Delta_\star, \mathbb{S} \vdash S \\ \vdots\ \Pi_{Gr\cup Sr} \\ \Omega_G; \mathbb{S} \vdash S \\ \vdots\ \Pi_{Cr\cup Sr} \\ \Omega; \mathbb{S} \vdash S\end{array}}
$$

first normal form

$$
\dfrac{\dfrac{\Pi_{Lr\cup\{\text{(L-Ax), (I-right)}\}}}{\Delta_\star, \mathbb{S} \vdash S}}{\begin{array}{c}\theta_\star, \Delta, \mathbb{S} \vdash S \\ \vdots\ \Pi_{\{(\multimap\text{-Merge})\}} \\ \Theta, \Delta, \mathbb{S} \vdash S \\ \vdots\ \Pi_{Gr\cup Sr} \\ \Omega_G; \mathbb{S} \vdash S \\ \vdots\ \Pi_{Cr\cup Sr} \\ \Omega; \mathbb{S} \vdash S\end{array}}\ (\multimap\text{-Spend})
$$

second normal form

We can reduce to only consider normal proofs, as stated by the following theorem.

Theorem 4.2 (normal form)**.** For any $\Omega, \mathbb{S}, S$, the initial sequent $\Omega; \mathbb{S} \vdash S$ is valid in CLNL* if and only if a normal proof Π exists for $\Omega; \mathbb{S} \vdash S$.

In the following we verify if a proof in the first normal form exists for an initial sequent, then we show how to reduce the second normal form case to the first one.

Solving with First Normal Form

The existence of the derivation $\Pi_{Cr\cup Sr}$ can be easily verified. Actually, such a derivation from $\Omega_G; \mathbb{S} \vdash S$ to $\Omega; \mathbb{S} \vdash S$ exists if and only if $\Omega \Vdash G\delta$ for each $G\delta \in \Omega_G$. We actually prove a stronger fact: we give a specific $\Omega_\star$ that subsumes all the possible cases. Formally

Lemma 4.1. A proof in the first normal form exists for $\Omega; \mathbb{S} \vdash S$ if and only a proof in the first normal form exists for $\Omega_\star; \mathbb{S} \vdash S$ where $\Omega_\star$ contains a single occurence of any $G\delta$ and $G\theta$ such that $\Omega \Vdash G\delta$ and $\Omega \Vdash G\theta$.

Since $\Vdash$ is deducibility in MIL, $\Omega_\star$ can be effectively computed using the decision procedure for intuitional propositional logic proposed in [74]. Since a proof in the first normal form contains no rule for linear implication, we can actually avoid considering $G\theta$ when computing $\Omega_\star$, since they will be discarded by (L-Weak) rules in any valid proof.

We consider now the proof obtained by composing the derivations $\Pi_{Gr \cup Sr}$ and $\Pi_{Lr \cup \{\text{(L-Ax), (I-right)}\}}$. Luckily an algorithm exists for deciding if such a proof exists in CLNL*.

Lemma 4.2. An always terminating algorithm exists that, given $\Omega_\star, \mathbb{S}$, and S, decides whether $\Omega_\star; \mathbb{S} \vdash S$ can be proved in CLNL* only using rules in $Gr \cup Sr \cup Lr \cup \{\text{(L-Ax), (I-right)}\}$.

This result derives from a similar one by Kanovich [80], which is stated for a computational fragment of linear logic that coincides with the sequents that we consider in $\Pi_{Gr \cup Sr}$ and $\Pi_{Lr \cup \{\text{(L-Ax), (I-right)}\}}$. In [80], Kanovich considers *simple products*, i.e., linear conjunctions of atomic predicates; *Horn-implications*, i.e., linear implications of simple products; and *!-Horn-implications*, i.e., Horn implications preceded by !. Moreover, he defines *!-Horn-sequents*, i.e., sequents with !-Horn-implications, Horn-implications and simple products as left parts and simple products as right part.

A translation from $\Omega_\star; \mathbb{S} \vdash S$ to !-Horn-sequents is trivially defined: $\mathbb{S}$ and S are simple products, while $\Omega_\star$ is translated by replacing G with ! (recall that $\Omega_\star$ does not contain contractual implications in the first normal form). Indeed, because of the restriction we have in $\Pi_{Gr \cup Sr}$, the rules applicable to propositions preceded by G are exactly to same of linear logic where G stands for !. Thus, the sequent $\Omega_\star; \mathbb{S} \vdash S$ is provable if and only if the !-Horn-sequent is valid. Finally, in [80], the problem of checking the validity of a !-Horn sequent (and thus also of our computational sequent) is reduced to reachability in Petri Nets, which can be decided using the algorithm proposed in [88].

Reducing Second to First Normal Form

For $\Pi_{Cr \cup Sr}$ we use the same approach of the first normal form.

We now consider the derivation from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega_G; \mathbb{S} \vdash S$. We build a multiset Ω'_G containing only linear propositions δ preceded by G, such that a derivation exists from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega_G; \mathbb{S} \vdash S$ if and only if a derivation exists from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega'_G; \mathbb{S} \vdash S$. This actually reduces the problem of finding a proof in the second normal form to finding one in the first normal form. Indeed, only a proof in the first normal form can exist for $\Omega'_G; \mathbb{S} \vdash S$, because Ω'_G contains no contractual linear implication, thus we cannot apply a ($\multimap\infty$-Spend rule). The construction of Ω'_G is based on a reduction to linear equations.

Let $\mathcal{L}_{\Omega_G} = \{\ell_1, \ldots \ell_p\}$ be the set of linear implications between atomic propositions $A \multimap A'$ appearing as terms in Ω_G. For every $G\delta \in \Omega_G$, let u_δ be a vector

of length p associating each index k with the number of occurrences of ℓ_k in δ; and for every $G\theta = G(\delta \multimap \delta') \in \Omega_G$, let u_θ and v_θ be vectors of length p associating each index k with the number of occurrences of ℓ_k in δ and δ' respectively. Moreover, let $u_{\Delta_\star}$ be a vector of length p associating each index k with the sum of the occurrences of ℓ_k in every $\delta \in \Delta_\star$. Finally, let A_{Ω_G} be the matrix with columns u_δ, B_{Ω_G} the matrix with columns u_θ, and C_{Ω_G} be the matrix with columns v_θ.

$$A_{\Omega_G} = \begin{bmatrix} | & | \\ u_{\delta_1} & u_{\delta_n} \\ | & | \end{bmatrix} \quad B_{\Omega_G} = \begin{bmatrix} | & | \\ u_{\theta_1} & u_{\theta_m} \\ | & | \end{bmatrix} \quad C_{\Omega_G} = \begin{bmatrix} | & | \\ v_{\theta_1} & v_{\theta_m} \\ | & | \end{bmatrix}$$

We proceed examining the derivation from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega_G; \mathbb{S} \vdash S$ using a bottom-up approach, showing a bind with the vectorial representation. Note that, because of the rules applicable in $\Pi_{Sr \cup Gr}$, the sequent $\Theta, \Delta, \mathbb{S} \vdash S$ is such that Θ may be composed by any arbitrary occurrence (possibly none) of θ, for $G\theta \in \Omega_G$, and Δ by any arbitrary occurrence (possibly none) of δ, for $G\delta \in \Omega_G$. From the only rule applicable in $\Pi_{\{(\multimap\text{-Merge})\}}$, we know that $\theta_\star = \delta_\star \multimap \delta'_\star$ is such that $\delta_\star$ is a linear conjunction of the left parts of $\theta \in \Theta$, and $\delta'_\star$ of the right parts of $\theta \in \Theta$. We also know that $\Delta_\star$ contains $\delta \in \Delta$, plus $\delta'_\star$. This means that the linear implications in $\Delta_\star$ are a linear combination of the ones in $G\delta \in \Omega_G$ and in the right parts of $G\theta \in \Omega_G$, with nonnegative integer coefficients. Formally,

$$u_{\Delta_\star} = \left[\begin{array}{c|c} A_{\Omega_G} & C_{\Omega_G} \end{array}\right] \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ z_1 \\ \vdots \\ z_m \end{bmatrix} \tag{4.10}$$

with $x_1, \dots x_n$ and $z_i, \dots z_m$ nonnegative integers.

Finally the rule ($\multimap$-Spend) can be applied if and only if $\delta_\star \subseteq \delta'_\star$, thus if and only if

$$\left[\begin{array}{c} C_{\Omega_G} - B_{\Omega_G} \end{array}\right] \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} \geq \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.11}$$

We can conclude that a derivation from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega_G; \mathbb{S} \vdash S$ exists if and only if all the previous conditions are met. We actually prove a more strong statement, that applies also to proofs in first normal forms.

Lemma 4.3. For any $\Omega_G, \Delta_\star, \mathbb{S}, S$, a derivation exists from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega_G; \mathbb{S} \vdash S$, in one of the following forms

$$\dfrac{\dfrac{\Delta_\star, \mathbb{S} \vdash S}{\theta, \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Spend})}{\vdots\ \Pi_{(\multimap\text{-Merge})}}$$

$$\begin{array}{ll} \Delta_\star, \mathbb{S} \vdash S & \Theta, \Delta, \mathbb{S} \vdash S \\ \vdots\ \Pi_{Gr \cup Sr} & \vdots\ \Pi_{Gr \cup Sr} \\ \Omega_G; \mathbb{S} \vdash S & \Omega_G; \mathbb{S} \vdash S \end{array}$$

if and only if $x_1, \dots x_n$ and $z_i, \dots z_m$ nonnegative integers exist such that formulas 4.10 and 4.11 hold.

Consider formula 4.11. For the Hilbert basis theorem [60], the set of nonnegative integer solutions can be expressed as

$$\begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} \\ H_{\Omega_G} \\ \\ \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_q \end{bmatrix}$$

with $y_1, \dots y_p$ nonnegative integers, and H can be computed using [17].

Thus, we build the following system precisely characterizing the solutions of formulas 4.10 and 4.11.

$$\left[\begin{array}{c|c} & \\ A_{\Omega_G} & C_{\Omega_G} \cdot H_{\Omega_G} \\ & \end{array}\right] \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ y_1 \\ \vdots \\ y_q \end{bmatrix}$$

Let D_{Ω_G} be the matrix above. We take a multiset Ω'_G containing a proposition $G\delta_v$ for each column v of D_{Ω_G} with

$$\delta_v = \bigotimes_{\ell_k \in \mathcal{L}_{\Omega_G}} \ell_k^{v_k} \tag{4.12}$$

where v_k is the value with index k in v, $\ell^0 = I$ and $\ell^{n+1} = \ell \otimes \ell^n$.

By construction, and Lemma C.15, the derivability of $\Omega_G; \mathbb{S} \vdash S$ is the same as the one of $\Omega'_G; \mathbb{S} \vdash S$. Formally:

Lemma 4.4. For any $\Omega_G, \Delta_\star, \mathbb{S}, S$, a derivation in the second normal form exists from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega_G; \mathbb{S} \vdash S$ if and only if a derivation in the first normal from exists from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega'_G; \mathbb{S} \vdash S$, with Ω'_G defined as in formula 4.12.

This allows us to reduce the problem of finding a proof in the second normal form to the one of finding a proof in the first normal form. Consequently, we can conclude hereof.

Theorem 4.3 (CLNL* decidablility)**.** An always terminating algorithm exists that decides if an initial sequent is valid in CLNL*.

4.3 Binding Layers: MuAC Formalization

We formalize now the semantics of the MuAC policy language, and define how the system evolves upon requests in terms of LNLC*.

4.3.1 MuAC Semantics

A *MuAC configuration* Σ associates users with their policies, i.e., with the appropriate set of rules. Formally $\Sigma : Usr \to 2^V$. The semantics $[\![\Sigma]\!]$ of a configuration Σ is a LNLC* theory, defined as follows.

$$[\![\Sigma]\!] = \bigcup_{usr \in Usr} \{[\![\nu]\!]_{usr} \mid \nu \in \Sigma(usr)\}$$

$$[\![r\colon P\ GiveLs]\!]_{usr} = \Lambda[u], \texttt{Requester}.\ [\![P]\!]_{usr} \to G([\![GiveLs]\!]_{usr} \multimap [\![\texttt{Gives}(\texttt{Requester}, r, \texttt{Me})]\!]_{usr})$$

$$[\![P]\!]_{usr} = \begin{cases} p([\![u_1]\!]_{usr}, \ldots, [\![u_i]\!]_{usr}) \wedge [\![P']\!]_{usr} & \text{if } P = p(u_1, \ldots, u_i)\ P' \\ \top & \text{if } P = \epsilon \end{cases}$$

$$[\![\texttt{Gives}(u, r', u')\ GiveLs]\!]_{usr} = (r'@[\![u']\!]_{usr} \multimap r'@[\![u]\!]_{usr}) \otimes [\![GiveLs]\!]_{usr}$$

$$[\![\texttt{Gives}(u, r', u')]\!]_{usr} = (r'@[\![u']\!]_{usr} \multimap r'@[\![u]\!]_{usr})$$

$$[\![\epsilon]\!]_{usr} = I$$

$$\text{with } [\![u]\!]_{usr} = \begin{cases} usr & \text{if } u = \texttt{Me} \\ u & \text{otherwise} \end{cases}$$

where $[u]$ is the set of user variables occurring in the expression $P\ GiveLs$, and predicates $r@u$ state that the user associated with the user variable u owns the resource of kind r. The symbol $\Lambda[u]$ stands for a restricted universal quantifier of $[u]$ over the *finite* set of users; it is just syntactic sugar that helps in compactly representing a finite conjunction of propositions ω.

Some comments are in order. The semantics $[\![\Sigma]\!]$ of Σ is a set of non-linear formulas, one for each rule ν of any user usr. Given a rule $r\colon P\ GiveLs$, a universally quantified non-linear formula $\Lambda[u].\omega \to G(\delta \multimap \delta')$ is added in $[\![\Sigma]\!]$ where ω encoding the non-linear conditions in P, δ the (linear) exchanges the user is requesting in return for r, and δ' the promise of usr to give r to the requester if the conditions are met. A MuAC statement $\texttt{Gives}(u, r', u')$ intuitively represents an exchange where u' gives a resource of kind r to u, i.e., $r@u' \multimap r@u$. As expected, if ν has many non-linear requirements, they are composed with $\wedge$; the same for linear requirements with $\otimes$. Finally, user variables u are bound to users in Usr by the finite universal quantifier Λ; with the exception of `Me`, which is interpreted as usr, the owner of the rule.

4.3.2 Formalizing MuAC System Evolution

The semantics $[\![\mathfrak{S}]\!]$ of a state $\mathfrak{S}$ of a MuAC system is a multiset of propositions $r@usr$, representing the current association of users with a number of resources of a given kind. The mapping between the function $\mathfrak{S}$ and the multiset $[\![\mathfrak{S}]\!]$ is trivially defined. We assume the semantics of a context Γ to be a MIL theory $[\![\Gamma]\!]$.

When a user *usr* asks for r the system is updated from $\mathfrak{S}_0$ to a new state $\mathfrak{S}_1$ if and only if the semantics of the MuAC configuration approves the update. Formally:

Definition 4.5 (MuAC System Semantics)**.** Give a MuAC system (Γ, Σ), its semantics is a system evolution such that

$$\Gamma, \Sigma \vDash \mathfrak{S}_0 \rightsquigarrow^{r@usr?} \mathfrak{S}_1$$

only if

$$[\![\Gamma]\!], [\![\Sigma]\!]; [\![\mathfrak{S}_0]\!] \vdash [\![\mathfrak{S}_1]\!]$$

Where $[\![\mathfrak{S}_1]\!] = r@usr \otimes S'$ for some S'.

Note that $[\![\Gamma]\!], [\![\Sigma]\!]; [\![\mathfrak{S}_0]\!] \vdash [\![\mathfrak{S}_1]\!]$ is actually an initial sequent

$$\Omega; \mathbb{S} \vdash S$$

where $\Omega = [\![\Gamma]\!] \cup [\![\Sigma]\!]$, $\mathbb{S} = [\![\mathfrak{S}_0]\!]$, and $S = [\![\mathfrak{S}_1]\!]$. Note also that $\mathfrak{S}_1$ stores the resource association that are not affected by the request, as well as the one due to the exchanges needed for granting *usr* the requested resource.

4.3.3 Examples

We review now the examples of subsection 4.1.2, giving the semantics to the policies of Alice, Bob and Carl and showing the proofs for their requests. The MuAC policies of the users are in Example 4.5. The semantics of the considered MuAC configuration Σ is the following multiset:

$$
\begin{array}{ll}
\{ & \\
\quad \Lambda u, u'.\, \top \to G((hw@u \multimap hw@Alice) \multimap\!\!\circ (sb@Alice \multimap sb@u')), & \textbf{A1} \\
\quad \Lambda u, u'.\, \top \to G((hp@u \multimap hp@Alice) \multimap\!\!\circ (sb@Alice \multimap sb@u')), & \textbf{A2} \\
\quad \Lambda u, u'.\, \top \to G((sb@u \multimap sb@Bob) \multimap\!\!\circ (lw@Bob \multimap lw@u')), & \textbf{B1} \\
\quad \Lambda u.\, \top \to G((sb@u \multimap sb@Bob) \multimap\!\!\circ (hp@Bob \multimap hp@u)), & \textbf{B2} \\
\quad \Lambda u.\, is\_paladin(u) \to G((hp@u \multimap hp@Bob) \multimap\!\!\circ (lw@Bob \multimap lw@u)), & \textbf{B3} \\
\quad \Lambda u, u'.\, \top \to G((lw@u \multimap lw@Carl) \multimap\!\!\circ (hw@Carl \multimap hw@u')), & \textbf{C1} \\
\quad \Lambda u.\, \top \to G((lw@u \multimap lw@Carl) \multimap\!\!\circ (hp@Carl \multimap hp@u)), & \textbf{C2} \\
\quad \Lambda u, u'.\, is\_paladin(u) \to G((sb@u' \multimap sb@u) \multimap\!\!\circ (hp@Carl \multimap hp@u')) & \textbf{C3} \\
\} &
\end{array}
$$

$$\frac{\Omega, \omega\{x \mapsto usr\} \Vdash \omega'}{\Omega, \Lambda x.\, \omega \Vdash \omega'}\,(\text{C-}\Lambda) \qquad \frac{\Omega, \omega\{x \mapsto usr\}; \Theta, \Delta, \mathbb{S} \vdash S}{\Omega, \Lambda x.\, \omega; \Theta, \Delta, \mathbb{S} \vdash S}\,(\text{L-}\Lambda)$$

Figure 4.6: Rules for Λ used in the examples.

We let the context Γ unspecified, and only assume that $[\![\Gamma]\!] \Vdash \mathit{is\_paladin}(\mathit{Bob})$ and $[\![\Gamma]\!] \Vdash \mathit{is\_paladin}(\mathit{Carl})$. Finally, the semantics S_0 of current state $\mathfrak{S}_0$ is the following multiset:

$$\{\mathit{sb@Alice}, \mathit{lw@Bob}, \mathit{hw@Carl}, \mathit{hp@Carl}\}.$$

We now revisit the proposed examples, showing how the system finds an agreement and thus a next state. For convenience, in the following we do not explicitly write $\Lambda[u].\, \omega$ as a long conjunction of serveral non-linear propositions, and we use the equivalent rules in Figure 4.6 instead. Moreover, in the proofs given in the Figures Figure 4.7, 4.8, 4.9, 4.10 we abbreviate *Alice*, *Bob*, and *Carl* with *A*, *B*, and *C* respectively. Also, we use double lines to represents multiple applications of deduction rules, and dashed lines when we omit trivial parts.

Example 4.8 (Direct exchange)**.** Consider Example 4.1. The request considered is *hp@Bob*?. The state $\mathfrak{S}_0$ is updated according to

$$\mathfrak{S}_0 \rightsquigarrow^{\mathit{hp@Bob?}} \mathfrak{S}_1 \quad \text{with} \quad [\![\mathfrak{S}_1]\!] = \mathit{hp@Bob} \otimes \mathit{sb@Alice} \otimes \mathit{hw@Carl} \otimes \mathit{lw@Carl}$$

using the LNLC* proof in Figure 4.7. Reading the proof bottom-up we can divide it into a non-linear part and a linear one.

The non-linear part: first we discard the MuAC rules not involved in the agreement; then we instantiate the quantified variables; we verify the non-linear conditions using the context Γ; finally, we remove the G on contracts, obtaining linear propositions.

The linear part: we merge left and right parts of contracts using (—∞-Merge); we verify that conditions are a subset of the promises results (—∞-Spend). Finally, the exchanges take places, and we compute the final result.

Example 4.9 (I pay for you)**.** The request considered in Example 4.2 is *sb@Bob*?. The state $\mathfrak{S}_0$ is updated according to

$$\mathfrak{S}_0 \rightsquigarrow^{\mathit{sb@Bob?}} \quad \text{with} \quad [\![\mathfrak{S}_1]\!] = \mathit{sb@Bob} \otimes \mathit{lw@Bob} \otimes \mathit{hp@Alice} \otimes \mathit{hw@Carl}$$

using the LNLC* proof in Figure 4.8. The structure of the proof is similar to the previous one, with the exception that *Carl* pays for *Bob*, which is permitted by the use of two variables in rule **A2**.

Example 4.10 (Circular Exchange)**.** The request considered in Example 4.3 is *hw@Alice*?. The state $\mathfrak{S}_0$ is updated according to

$$\mathfrak{S}_0 \rightsquigarrow^{\mathit{hw@Alice?}} \mathfrak{S}_1 \quad \text{with} \quad [\![\mathfrak{S}_1]\!] = \mathit{hw@Alice} \otimes \mathit{sb@Bob} \otimes \mathit{lw@Carl} \otimes \mathit{hp@Carl}$$

using the LNLC* proof in Figure 4.9. The only significant difference is that the rules to consider are three, thus (—∞-Merge) is applied twice.

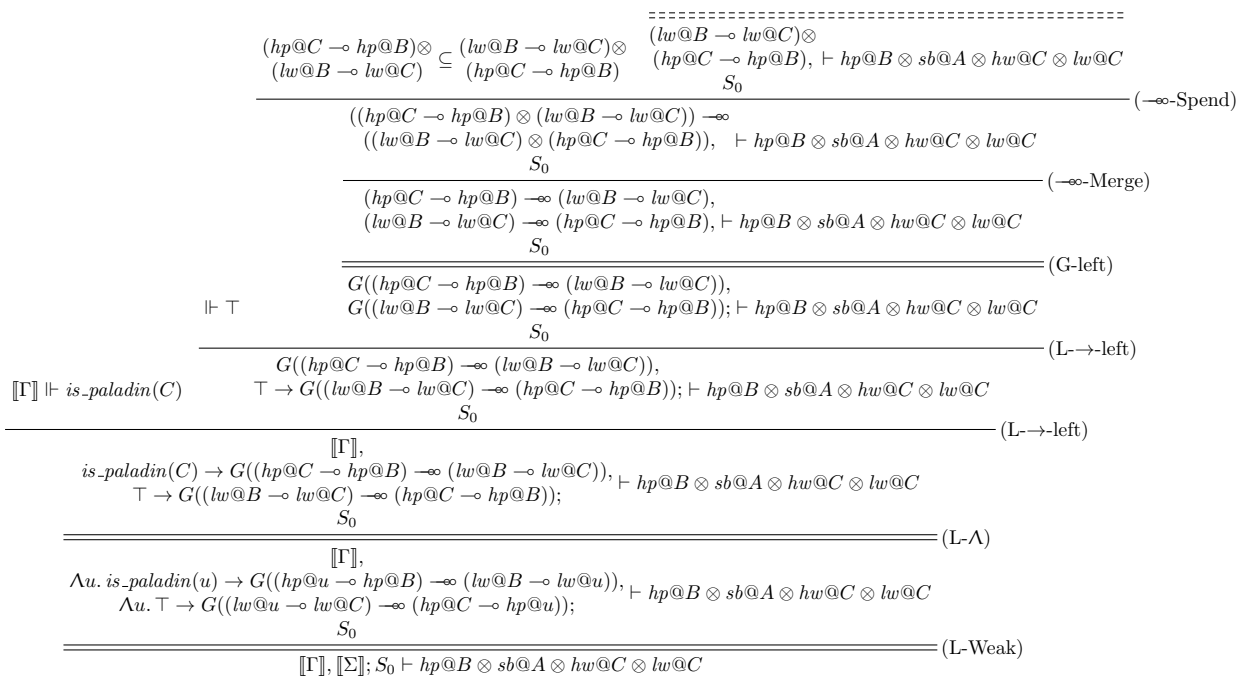


Figure 4.7: LNLC* derviation for Example 4.1.

Example 4.11 (Resource Supplier)**.** The request considered in Example 4.4 is $hp@Alice?$. The state $\mathfrak{S}_0$ is updated according to

$$\mathfrak{S}_0 \rightsquigarrow^{hp@Alice?} \mathfrak{S}_1 \quad \text{with} \quad [\![\mathfrak{S}_1]\!] = hp@Alice \otimes sb@Bob \otimes lw@Carl \otimes hw@Carl$$

using the LNLC* proof in Figure 4.10. The differences here are that we consider two pairs of promises: one between *Bob* and *Carl* and one between *Alice* and *Bob*, and that in the linear part we have two linear implications ($hp@C \multimap hp@B$) and ($hp@B \multimap hp@A$) for shipping the *hp* resource to *Alice*.

4.4 Implementing the MuAC System

In this section, we address the problem of implementing MuAC. First, we give an algorithmical characterization of the evolution of the MuAC system, i.e., how it evaluates the requests. Note that LNLC* dedidability is not enough, since the sequent to use for computing the next state is not fixed. Finally, we present a possible implementation of MuAC as a smart contract, where the heavier part of the computation is performed off-chain and the contract is only used to grant the correctness.

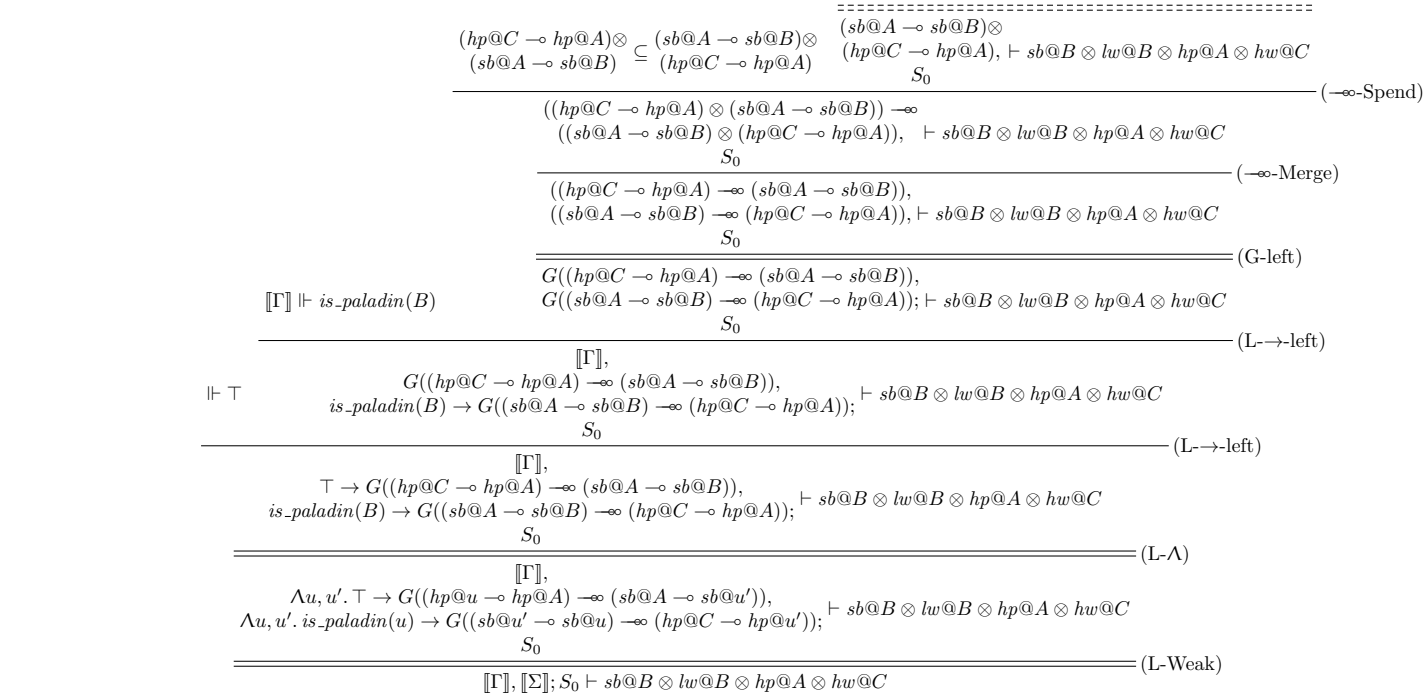


Figure 4.8: LNLC* derviation for Example 4.2.

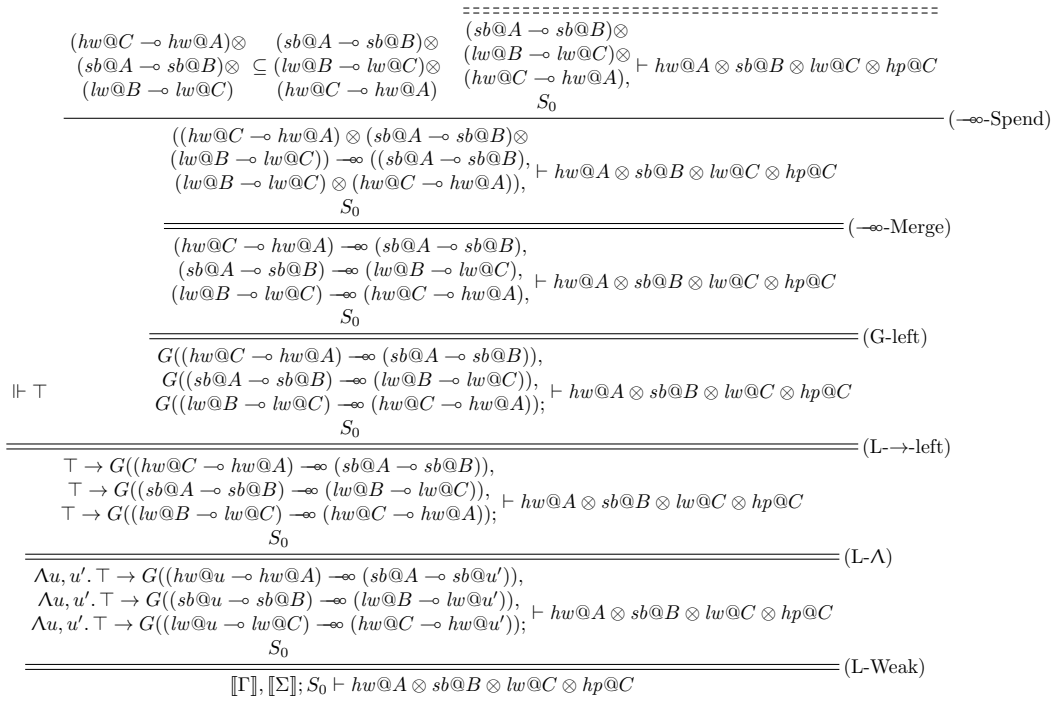


Figure 4.9: LNLC* derviation for Example 4.3.

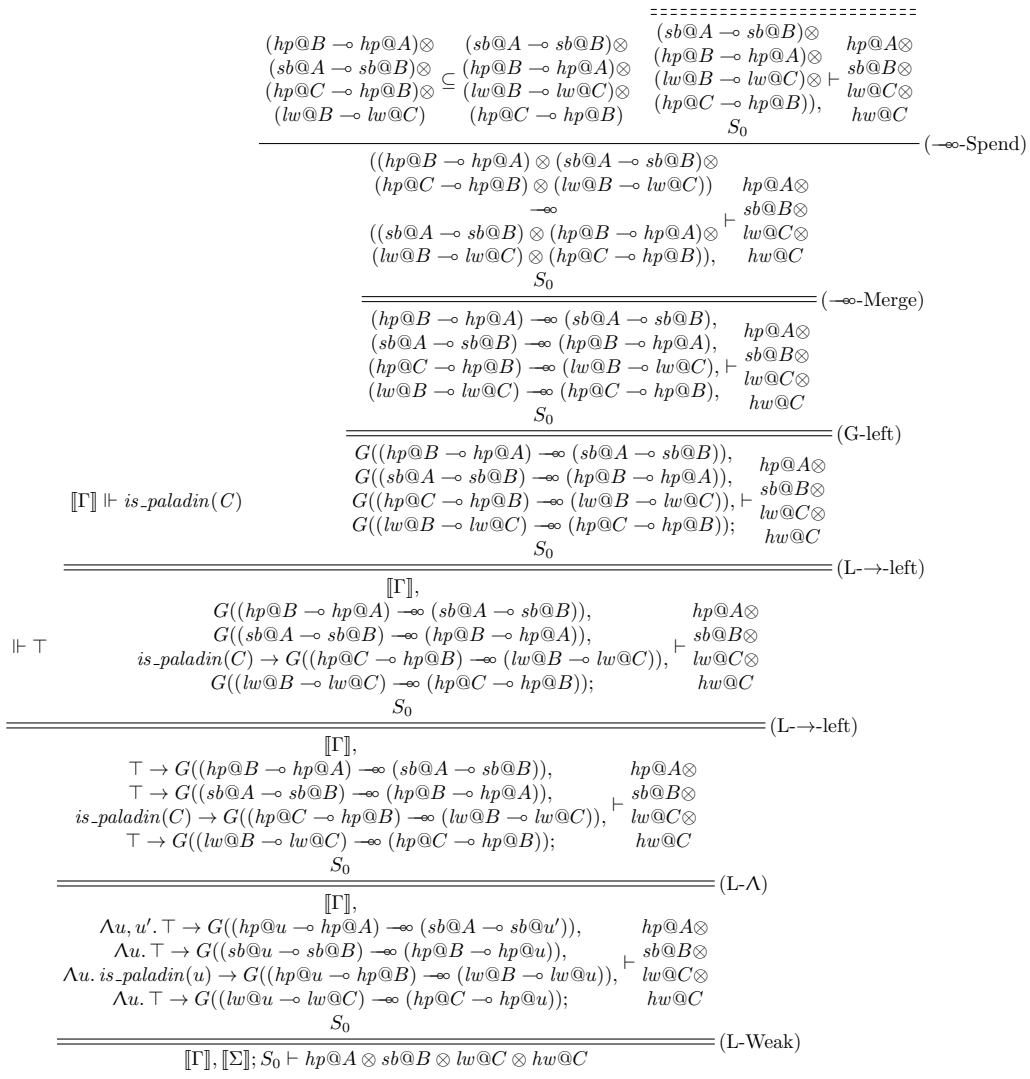


Figure 4.10: LNLC* derviation for Example 4.4.

4.4.1 Computing System Evolution

Given a MuAC system (Γ, Σ) in a state $\mathfrak{S}_0$, and given a request $r@usr?$, the problem is either finding a next state $\mathfrak{S}_1$ such that

$$\Gamma, \Sigma \vDash \mathfrak{S}_0 \rightsquigarrow^{r@usr?} \mathfrak{S}_1$$

or prove that there are none. This reduces to finding if

$$[\![\Gamma]\!], [\![\Sigma]\!]; [\![\mathfrak{S}_0]\!] \vdash r@usr \otimes S'$$

holds for some S'. Given S' this can be done using the method of subsection 4.2.3. The problem we have to address is to determine which S' to consider among the possibly infinite states.

Define the *quantity* $q(\varphi)$ of a linear formula φ as

$$q(\varphi) = \begin{cases} 1 & \text{if } \varphi = r@u \\ q(\varphi') + q(\varphi'') & \text{if } \varphi = \varphi' \otimes \varphi'' \\ 0 & \text{otherwise} \end{cases}$$

Let $q(\Phi)$ be the quantity of a linear theory Φ, defined as the sum of $q(\varphi)$ for $\varphi \in \Phi$. Intuitively, $q(\Phi)$ is the number of atomic linear predicates that appear in Φ not bound by logical connectives other then $\otimes$. We have the following property by trivial rule induction.

Lemma 4.5. For each $\Omega, \mathbb{S}, S$, if $\Omega; \mathbb{S} \vdash S$, then $q(\mathbb{S}) = q(S)$

We define the atomic linear subformulas of a formula φ or ω as follows:

$$\begin{aligned} &asub(r@u) = \{r@u\} \\ &asub(\varphi \star \varphi') = asub(\omega) \cup asub(\omega') \quad \text{with } \star \in \{\otimes, \multimap, \mathrel{-\!\!\circ}, \wedge, \to\} \\ &asub(G\varphi) = asub(\varphi) \end{aligned}$$

We homomorphically extend this definition to multisets of linear and non-linear predicates, and we prove the following result by trivial rule induction.

Lemma 4.6. For each $\Omega, \mathbb{S}, S$, if $\Omega; \mathbb{S} \vdash S$, then $asub(S) \subseteq asub(\Omega) \cup asub(\mathbb{S})$.

Lemma 4.5 and 4.6 ensure that it suffices to consider only a finite number of candidates for computing the next state. This, plus the LNLC* decidibility allow us to state the following theorem.

Theorem 4.4 (Computability)**.** There exists an algorithm that, given a MuAC system (Γ, Σ) in a state $\mathfrak{S}_1$, and given a request $r@usr?$, finds all the states $\mathfrak{S}_1$ such that $\Gamma, \Sigma \vDash \mathfrak{S}_0 \rightsquigarrow^{r@u?} \mathfrak{S}_1$.

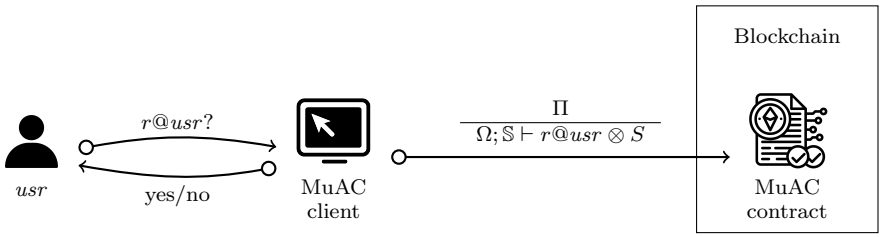


Figure 4.11: MuAC implementation on Ethereum.

4.4.2 MuAC as a Smart Contract

We show how a MuAC system can be exploited for exchanging NFTs. We propose a strategy where a smart contract stores the association between users and resources, and an off-chain application serves user requests. The main idea is to let the client perform the most expensive part of the computation, whereas only a small verification is handled in the blockchain.

In the following, we first give an overview of the workflow for serving an access request, we then sketch a possible implementation.

Overview

The proposed workflow for serving a request, depicted in Figure 4.11, is as follows:

1. The user asks for a resource using its client;
2. Using the procedure of subsection 4.2.3, the client derives a LNLC* proof if any, and denies the request otherwise;
3. The client proposes a change of the state to the smart contracts, attaching the LNLC* proof;
4. The smart contract checks the proof correctness, updating the state if this is the case (thus serving the request), or else signaling an error.

Note that verifying the LNLC* proof is linear on the number of rules of the proof. Reducing the cost of the computation performed by the contract is critical, because in blockchains like Etherum every executed instruction is paid by the requester using a in-block currency (called Gas in Ethereum). With the proposed workflow, all but a linear portion of the computation is performed off-chain. Nevertheless, the system guarantee transparency and correctness of the exchanges. The rules for accessing the resources are in clear on the contract, whose execution is ensured by the blockchain.

Implementation

For convenience, we abstractly model the actual implementation of smart contracts and NFT in real world blockchains, and we do not address the creation of NFTs.

We consider three kinds of addresses: *user accounts*, *smart contracts* and NFTs. A NFT is associated with an owner, that may be a user account or a smart contract. A smart contract has a set of fields, i.e., its internal state, and exposes a set of functions, that can be called by users or other smart contracts. Smart contracts and users interact through messages that are used for calling functions and transfer NFTs. Required fields of a message are the *sender* and *destination* addresses (of users or smart contracts). Optionally, the message contains a *function* field, with the name of the function to call, a *parameters* field for the actual parameters of the called function, and a *token* field, containing the NFT. If the receiver is a smart contract, the code of the function to call is executed, changing the internal state of the contract and causing the smart contract to send messages in turn. If the message contains a NFT, the receiver becomes its owner.

The pseudocode of the MuAC smart contract is in Figure 4.12a. The internal state consists in a field $\mathfrak{S}$, i.e., the state of the MuAC system, a field Σ, i.e., the MuAC configuration, and Γ, i.e., the context. For taking part in MuAC exchanges, the player transfer the NFTs of the resources to exchange to the MuAC smart contract using function `add_resource`. The message sets the owner of the token to the MuAC contract, and cause the update of $\mathfrak{S}$. At any moment, users can have their resources back with function `recover_resource`. The function checks that in $\mathfrak{S}$ the resource is actually associated with the requester, if this is the case $\mathfrak{S}$ is updated by removing the resource, and a message is sent to the user carrying the token, otherwise the computation fails and the state is unchanged. Users can also change their policies with the function `update_policy`. Finally, a user can propose an evolution of the state of the MuAC system, providing a LNLC* proof Π. If Π is valid for $\Omega; \mathbb{S} \vdash S$, and if $\mathbb{S} = [\![\mathfrak{S}]\!]$, $\Omega = [\![\Gamma]\!] \cup [\![\Sigma]\!]$, then the state $\mathfrak{S}_1$, taken from Π with function `next_state`, updates the MuAC state.

The pseudocode of the MuAC client is in Figure 4.12b. When receiving a request from a user *usr* for a given resource kind r, the client recovers the parameters $\Gamma, \mathfrak{S}, \Sigma$ from the smart contract. Then it uses the procedure of subsection 4.4.1 to find a proof Π for a transition to a next state satisfying the request. If such a proof Π exists, a message is built and sent to the blockchain through the user account. The message has the MuAC contract address as destination, `system_evolve` as function to call, and Π as a parameter. We assume the function `BCsend` to be in a library for interacting with the blockchain, configured with the access data of *usr*.

Example 4.12. Consider our running example, and assume the users have initialized the state calling `resource_add` with the addresses of their resources. To make the request of Example 4.1, Bob calls his client asking for *hp*. The client recovers the parameters and finds the proof of Figure 4.7. Then, it sends

```
Contract MuAC
    𝔊 : user_address → 2^resource_address
    Σ : user_address → MuAC_policy
    Γ : MLL theory

    function add_resource()
        𝔊[msg.sender] ← 𝔊[msg.sender] ∪ {msg.token}

    function recover_resource(resource)
        Require(resource ∈ 𝔊[msg.sender])
        𝔊[msg.sender] ← 𝔊[msg.sender] \ {resource}
        send(msg.sender,resource)
        resource.transfer(msg.sender)

    function update_policy(V)
        Σ[msg.sender] ← V

    function system_evolve(Π)
        Require(Verify(Π))
        𝔊 ← next_state(Π)
```

(a) MuAC contract pseudo-code.

```
function serve_request(r, usr)
    Γ ← take_context_from_contract()
    𝔊 ← take_state_from_contract()
    Σ ← take_configuration_from_contract()
    Π ← find_next_state(Γ, Σ, 𝔊, r, usr)
    if(Π = null) then
        print "Error: request denied"
    else
        message ← empty_message
        message.function ← MuAC.evolve
        message.parameters ← Π
        BCsend(message)
```

(b) Algorithm of the MuAC client.

Figure 4.12: Implementation of a MuAC system on a blockchain.

a message to the contract that verifies that the proof is valid, and the state is updated as presented in Example 4.8. Now the user owns a resource of kind *hp* as requested.

4.4.3 Context of Application

The choice of an online game as a running example is mainly motivated by its simplicity. So far we focused on a simplified representation of online games, glossing on the dynamicity of such applications. Hereafter, we discuss the dynamicity-related details, and how they can be addressed by MuAC. One problem that may arise is that the need for certain types of resources can change frequently, e.g., when the desired resource is obtained. This implies that additional efforts might be required from the user to maintain their policies up to date. A second problem is that the resources that are exchanged using a MuAC smart contract, once sent to the contract, are not immediately available to the original owner, that has to recover them first (see Figure 4.12a).

These limitations are not significant in scenarios where the policies as well as the available resources do not change frequently over time. The exchange of infinite or reusable resources (discussed in Section 4.5) is free from problems

related to the dynamicity of available resources. Thus MuAC can be used, e.g., for file sharing in peer-to-peer contexts like torrents.

We focus now on finite resources. In subsection 4.4.2 we show how MuAC can be applied inside blockchains for exchanging ownership over NFT. This approach can be extended for exchanging smart contracts in general. Blockchains are not usually adequate for dynamic environments. However several works try to solve this limitation by resorting to off-chain applications. Their usage has been proposed, e.g., for managing real-world assets according to the sharing economy paradigm [129]. Either by using a blockchain, or resorting to more traditional approaches, MuAC naturally encodes mutuality policies for sharing environments.

In the matter of the dynamicity of policies, policy updates may be delegated to client applications that, e.g., remove a rule for a resource when a number of similar ones have been received. In Section 4.6 we discuss a possible extension of MuAC for automatically adjusting the policies based on a game-theoretic approach. When MuAC is implemented as a blockchain smart contract, policies may be automatically updated together with the system state when the contract is called. As in the standard case, a off-chain application may be used to perform the expensive computation (finding the best policy update), and the contract only has to check the correctness of the result.

4.5 Dealing with Reusable Resources

In this section we see how to adapt MuAC when the resources are reusable, i.e. the asset is still available to the owner after he allows access to it. The only thing we need to do is to slightly change the semantics of MuAC configurations, and adapt the system evolution accordingly.

4.5.1 MuAC Semantics Revisited

Since resource are reusable, there is no need of changing the owner of them when a request is served. The state $\mathfrak{S}$ of a MuAC system associates users and resource of other users with *accept* or *deny* results. Formally, $[\![\mathfrak{S}]\!]$ is a conjunction of atomic propositions $allow(usr, r, usr')$ stating that usr can access the resources of kind r owned by usr'.

The semantics of MuAC languages for reusable resource is similar to the previous one of subsection 4.3.1.

$$[\![\Sigma]\!] = \bigcup_{usr \in Usr} \{[\![\nu]\!]_{usr} \mid \nu \in \Sigma(usr)\}$$

$$[\![r\colon P\ GiveLs]\!]_{usr} = \Lambda[u], \texttt{Requester}.$$
$$[\![P]\!]_{usr} \to G([\![GiveLs]\!]_{usr} \multimap [\![\texttt{Gives}(\texttt{Requester}, r, \texttt{Me})]\!]_{usr})$$

$$[\![P]\!]_{usr} = \begin{cases} p([\![u_1]\!]_{usr}, \ldots, [\![u_i]\!]_{usr}) \wedge [\![P']\!]_{usr} & \text{if } P = p(u_1, \ldots, u_i)\ P' \\ \top & \text{if } P = \epsilon \end{cases}$$

$$
\begin{aligned}
&[\![\texttt{Gives}(u, r', u'), GiveLs]\!]_{usr} = \\
&\qquad (I \multimap allow([\![u]\!]_{usr}, r', [\![u']\!]_{usr})) \otimes [\![GiveLs]\!]_{usr} \\
&[\![\texttt{Gives}(u, r', u')]\!]_{usr} = (I \multimap allow([\![u]\!]_{usr}, r', [\![u']\!]_{usr})) \\
&[\![\epsilon]\!]_{usr} = I \\
&\text{with } [\![u]\!]_{usr} = \begin{cases} usr & \text{if } u = \texttt{Me} \\ u & \text{otherwise} \end{cases}
\end{aligned}
$$

The only difference is the interpretation of the *Gives* expression. Recall that, in the formulation of subsection 4.3.1, *Gives* is interpreted as a linear implication that creates a new association of the resource of kind r with usr and removes the association with the original owner usr. For reusable resources, a proposition stating that the usr is allowed to access the resource suffices, thus we simply use $I \multimap allow(usr, r, usr')$.

4.5.2 System Evolution Revisited

When a user usr asks for a resource of kind r, if the request can be served, the system is updated from S to a new state in which usr can access a resource of kind r, in symbols:

$$\Gamma, \Sigma \vDash \mathfrak{S}_0 \rightsquigarrow^{r@usr?} \mathfrak{S}_1$$

This update is feasible if and only if

$$[\![\Gamma]\!], [\![\Sigma]\!]; [\![\mathfrak{S}_0]\!] \vdash [\![\mathfrak{S}_1]\!] \quad \text{with} \quad [\![\mathfrak{S}_1]\!] = allow(usr, r, usr') \otimes S'$$

for some usr' and S'.

As for the previous formulation, this is actually an initial sequent

$$\Omega; \mathbb{S} \vdash S$$

where $\Omega = [\![\Gamma]\!] \cup [\![\Sigma]\!]$, $\mathbb{S} = [\![\mathfrak{S}_0]\!]$, and $S = [\![\mathfrak{S}_1]\!]$. Note that $\mathfrak{S}_1$ stores all the permissions in $\mathfrak{S}_0$ plus the ones given in return for the requested permission.

4.5.3 Discussion About Linearity

One may ask why linearity is needed when resources are reusable. The system can indeed be rewritten using the non-linear logic proposed in [20], but using linear logic we grant mutuality even when users change their policies. We show that this guarantee applies through the following example.

Example 4.13. Consider two hospitals storing anonymized medical data. Assume that hospital A stores flu related data, whereas hospital B stores covid related data, and that both decide to share their data only with hospitals that share their data in return. Consider a MuAc system where the only kinds of resources are *flu* and *covid*. The MuAC policy of hospital A would be the following rule.

```
flu: Gives(Me, covid, Requester)
```

And the one of hospital B would be:

```
covid: Gives(Me, flu, Requester)
```

If hospital A makes a request for *covid* data, two new permissions are added to the state of the MuAC system: $allow(A, covid, B)$ and $allow(B, flu, A)$. If in the future, hospital B makes a requests for *flu* data, hospital A cannot refuse to share its data. Recall that users cannot directly change the state of the MuAC system, but only their policies. Even if hospital A change its policy to always deny, only further exchanges of permissions are refused, but the past agreements must be honored, since the state has been updated. Indeed, when the request for *flu* data arrives from hospital B, the policies are not even checked, the relevant permission is already in the sate of the MuAC system.

4.6 Related Work

Here we only consider discretionary access control [115] because it is a natural choice in distributed cooperative setting, where users individually decide the policies for the resources they own. In this environment, a main issue is the combination of individual policies. To the best of our knowledge, no proposals address mutuality, but only focus on the resolution of conflicts [29, 46, 100]. In the restricted, yet widespread distributed world of social networks, mutuality plays a prominent role, but is scarcely regulated. A remarkable exception is [118] that permits defining mutual access control policies. This is done by introducing a new grant, called *mutual*, besides the usual *accept* and *deny*. Suppose that an access request from user A to resource r of B evaluates to *mutual*. Intuitively, the request is served if and only if a request from B for a *similar* resource r' of A will evaluate to *accept* or *mutual*. Similarity is fixed once and for all, and is not user-defined. A first difference with our proposal is that mutuality is defined through explicit constraints in the body of the rules, so allowing users to define their own notion of similarity. In addition, mutuality in *MuAC* may involve many users, as in Example 4.3, and we target also finite resources.

Mutuality plays a main role also in trust negotiation, i.e., a process that permits a safe interaction between two parties that do not trust each other [82]. The idea is to run a multi-round protocol where the parties exchange some pieces of private information (*credentials*) so as to increase their mutual trust. Also in this setting, each party defines an individual policy specifying the conditions that the other party must satisfy in turn to obtain credentials. The overall goal is to balance the disclosure of information and the mutual benefit gained by each party. Logical languages for specifying trust policies have been proposed, e.g., Cassandra [24] and SecPal4P [23]. However, the main difference with our proposal is that these are based on classical logic, and thus circular conditions do not lead to an agreement.

Some of the works that focus on conflict resolution [72, 128, 117, 104] are based on game theory. They assume that different users may be in different

relation with a resource in a social network. For example, one user may upload a picture, another may appear in it, and a third user may spread the picture by reposting it. Each one have an interest for the picture to be accessible or hidden for different sets of users. Conflicts may arise between the preferences of different users, and the optimal solution can be defined, e.g., as the maximum point of a social utility function or as a Nash equilibrium. For example, in [72] Hu et al. propose a model of the *privacy risk*, i.e., the risk implied by granting access to sensible data, and of the *sharing loss*, i.e., the damage obtained by denying access. A multiparty control game represents the strategic reasoning of the users adjusting their policies to maximize their own benefit. The authors show that a unique Nash equilibrium exists, and propose two algorithms that converge to such equilibrium in a few iterations in two different conditions: synchronous and non-synchronous adjustments, respectively. They also experimentally study the gap between the results of game theoretic approaches, that assume perfect reasoners, and the real human behavior.

Xiao et al. [128] make different assumptions. They notice that in social networks users are not entirely selfish. Since the connections on a social network are often based on meaningful human relationship like friendship or family, users usually care not only on their feeling about sharing something, but also on the feelings of others. The authors call it *peer effect*. The ideal policy of a user thus depends on his preference and a weighed sum of the preferences of others. They assume the users initially rank the access control policies according to their own preference, and later adjust them based on the peer effect. The policies of their peer also changes, and this continues until an equilibrium is reached. Xiao et al. show that a unique equilibrium exist, and is automatically computed by CAPE, the mechanism they propose. Once the equilibrium is reached, a combination algorithm can be used to take a decision for multi-owned resources (e.g. full-consensus, one-override, majority).

Like in the game-theoretic approach, we assume users to be fully rational *perfect reasoners*, but our target is not about multiple ownership, every resource has a single owner in MuAC. For this reason, the desire of the other users to gain access to the resource is not to be compared with the owner desire, that cannot be violated. Of course, in MuAC every user would be happy to access the resources without giving anything in return, but instead of asking users to rank the possible access policies, we allow users to express under which condition a compromise is satisfying. When an access request is performed, it is often the case that more different agreements can be reached, and a game-theoretic approach could be applied in conjunction with MuAC for deciding which agreement should be preferred. For example, if Bob makes a request to Alice for an apple, and Alice is fine with giving apples for either grapes or cheese, both an exchange of apples for grapes and an exchange of apples for cheese are possible. MuAC does not currently allow expressing preferences over the acceptable exchanges, and this limitation can be overcome by asking users to rank their rules. Conflicts may arise between Bob, preferring to give cheese, and Alice preferring to receive grapes. A notion of social utility may be defined, as well as an equilibrium for rank adjusting, but in contrast with the previous

proposals, only agreements acceptable for all the users would be considered.

The logical aspects of our work is based on the proposal of Bartoletti et al. [20], who first proposed PCL, a logic for modeling contractual reasoning. Our operator $\multimap$ is actually a linear version of their $\twoheadrightarrow$. The main difference with PCL is that from $p \twoheadrightarrow p', p' \twoheadrightarrow p$ you can either derive p, p', or $p \wedge p'$, but in our system only the whole pair $p \otimes p'$ can be derived from $p \multimap p', p' \multimap p$. In addition, our logic mixes linear and non-linear terms. We have applied it in the context of access control, also proposing an implementation on block chains. A second work our is heavily based on is [25], where a mixed linear non-linear logic is proposed. We extended this logic with $\multimap$ and rules for reasoning about contracts and promises.

4.7 Conclusions and future work

In this chapter, we have proposed a new kind of access control (the high level) and have shown how it the can be implemented using non-standard logic (the low level). The high level is MuAC, an access control for a distributed collaborative environment. Its main feature concerns expressivity, as it permits to define and handle access requests that demand mutual agreement between several users, both in case of reusable and non reusable resources. In the low level, we resolve the circularity that may arise when dealing with mutuality by defining the new operator $\multimap$ that models promises in contracts. Moreover, we have embedded $\multimap$ in mixed linear non-linear logic, showing that a computational fragment is adequate for representing the semantics of MuAC policies. As an instance of the two-layer approach, the user only deals with the abstract specification of the desired behaviour, while all the details are managed by the low level that resorts to logic for deciding access requests. Finally, we have proposed an implementation of MuAC as a smart contract, showing that most of the computation can be performed off-chain.

Future Work Future work can follow two distinct paths. The first covers LNLC that we plan to further investigates. The rules for the computational fragment are proved to be correct, but not complete. Having a complete characterization of the fragment without loosing decidability is one of our main research topics for the future. We also plan to investigate the possibility of using more expressive propositions where formulas having $\multimap$ may appear inside other formulas with $\multimap$. This would allow to state promises like "if you start trading r for r', I will start trading r'' for r'''." We are currently working on a model-theoretic semantics for the computational fragment of LNLC, and we hope to extend it for the complete logic. Intuition suggests that contractual reasoning in CMILL implies a sort of circular reasoning about derivations, e.g., from $\varphi \multimap \varphi'$ one derives φ' if φ derives from $\varphi' \multimap \varphi$, and vice-versa. Although the derivation rules we propose solve this circularity in a way that fits well with the intuitive meaning of contracts (see subsection 4.2.1), we will also investigate alternative characterizations based on greatest fixed-points.

The second path is about extending the MuAC policy language. We plan to integrate numerical theories in MuAC, thus allowing the user to speak about the number of resources. Thus would foster the usage for selling resources and exchanging currencies. For the time being, MuAC has positive `Gives` grants only. Also having negative rules will be of great interest, which also require to address the resolution of potential conflicts. Another extension is allowing rules in which a user specifies resources that must not be shared. For example, Alice permits Bob to access her pictures, provided that Bob does not share anything with Charlie. This kind of negative requirement might be a first step towards the definition of policies regulating conflicts of interest.

Chapter 5

Conclusions

We have proposed an approach for solving the common problems that may arise because of poorly managed interaction between specifications and implementation of access control policies. these two tasks require users to switch from one level of abstraction to another, often changing language and tools. We have proposed a general approach for security engineers to interact with access control systems at different abstraction layers for configuring, updating and verifying system behaviour. Our two-layers approach guarantees that the two representations of the system are coherent, granting a correct communication between high level specifications and low level executable configurations. We have applied it in three different contexts, proposing solutions that are based on their constraints and peculiarities. In particular, we have addressed networks, operating systems and collaborative environments. For each of them we have proposed a formal model of the low level of the executable configurations and we have designed a high level inspired by the needs of policy designers.

For network security we have considered firewalls. At the high level, the specifications of a firewall are represented as a function over IP packets. We have proposed a SQL-like query language for modifying and verifying the specifications in a handy manner, abstracting away from low level details like shadowing, tags and the limitations of packet matching. For the low level we have proposed a formalization based on the encoding of `iptables`, `pf` and `ipfw` in IFCL, a common intermediate language that subsumes them all. The coherence between the functional representation of the firewall and its actual configuration is obtained by a two-way translation, namely a compiler from functions to configurations and a decompiler. We also investigated the expressive power of the considered languages, showing two hierarchy, one that consider tag systems, the other focusing only on the basic and most used features. We found out that some functions cannot be implemented in some specific language. We have presented and experimentally validated two tools: FWS, that implements the two-way translation and also supports the administrator in varius tasks; and F2F, a tool that checks if a given function is expressible in the target firewall language, also detecting if tags are needed.

For system security, we have targeted SELinux CIL configurations. At the high level, we represent the system as information flows between OS entities. Thus, we have proposed IFL, a domain specific language for defining fine grained information flow requirements (including confidentiality, integrity and non-transitive properties). IFL expresses both functional requirements, i.e., which permissions must be granted to users for performing their authorized tasks, and security requirements, i.e., information flows to forbid because possibly dangerous. We have proposed a formal semantics of SELinux CIL configurations, that we have empirically validated discovering lots of counterintuitive corner cases and disagreements between the documentation and the compiler. We have presented IFCIL, an extension of CIL where IFL requirements are first class citizens, and a verification procedure for granting coherence between the information flow specifications and the actual configuration where permissions are explicitly listed for each entity. The tool IFCILverif implements the verification procedure by statically checking that all the requirements are met in a IFCIL configuration.

Finally, we have applied the two-layer approach to collaborative environments, in which the users interact by sharing or exchanging assets. We have addressed the problem of finding mutually advantageous agreements between users provided a specification of their promises and requests. More in details, we have proposed MuAC, an high level language with which each user can express what they want in return for allowing access to their assets. The low level language is based on a non-standard logical theory, and the evaluation of access requests rely on logical deductions. We have addressed both the case of infinite or reusable assets, where the asset is still available to the owner after he allows access to it, and the case of finite resources that are consumed when exchanged. We have established a binding between the high level and the logical low level through a compilation from MuAC policies to logical theories that also gives semantics to MuAC. The proposed implementation evaluates the access requests by changing ownership of the resources or updating an access control matrix, driving the system to behave correctly. Finally, we have also proposed a realization of MuAC as a blockchain smart contract where most of the computation is performed off-chain.

Future Work

In network security, we will further extend our approach to other firewalls like Cisco-IOS, and to other paradigms like SDN. The major difficulty in this case arises because of the high dynamicity of SDNs, while our proposal focuses on legacy networks and devices that are essentially static. It would be very interesting to extend our approach to deal with networks with more than one firewall. In systems we plan to extend our tool 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. In collaborative environments we will extend the MuAC policy language for dealing with currencies, and to allowing rules in which a user specifies resources that must not be shared, like

in conflicts of interest.

A promising line of research is about incrementality and compositionality, i.e., only propagates the modification from high to low level representations, without recompiling the whole policy. In translations-based solutions, this would allow to maintain properties of the low level configuration, e.g. logs and the internal structure of the rulesets in firewalls. Also in SELinux this is critical for integrating 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.

There are several directions for future work that aim at fostering the adoption of our tools by practitioners. In F2F and IFCILverif we will provide more friendly diagnostics and suggestions for fixing configurations. In IFCILverif we also plan to support configurations partly written in the kernel policy language and partly written in CIL, as this is common practice [6]. We will also give a prototypical implementation of MuAC, and we will investigate efficient solutions for evaluating access requests.

Future work also includes considering different access control systems. We think that our two-layers approach can be applied with two-way translation based solutions to ABAC, because the low level intricacy are similar to the ones of firewalls. We will also investigate the implementation of highly expressive, logically-based access control systems using approaches similar to the one of MuAC.

Bibliography

[1] AWS. https://aws.amazon.com.

[2] extensible access control markup language (xacml) version 3.0. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[3] Openstack. https://www.openstack.org.

[4] openWRT project. https://openwrt.org.

[5] SELinux IFCIL tool. https://sites.google.com/view/ifcilpaper.

[6] sepolicy. https://android.googlesource.com/platform/system/sepolicy/.

[7] Oasis extensible access control markup language, 2013. http://xacmlinfo.org/category/xacml-3-0/.

[8] Shorewall. http://www.shorewall.net/, 2014.

[9] The IPFW Firewall. https://www.freebsd.org/doc/handbook/firewalls-ipfw.html, 2017.

[10] Netfilter. https://www.netfilter.org/, 2019.

[11] Packet Filter (PF). https://www.openbsd.org/faq/pf/, 2019.

[12] F2F tool. https://github.com/lceragioli/F2F, 2021.

[13] FWS tool. https://github.com/secgroup/fws, 2021.

[14] P. Adão, C. Bozzato, G. Dei Rossi, R. Focardi, and F. L. Luccio. Mignis: A Semantic Based Tool for Firewall Configuration. In *proc. of the 27th IEEE CSF*, pages 351–365, 2014.

[15] P. Adão, R. Focardi, J. D. Guttman, and F. L. Luccio. Localizing firewall security policies. In *proc. of the 29th IEEE CSF, Lisbon, Portugal, June 27 - July 1*, pages 194–209, 2016.

[16] T. Ahmed, R. Sandhu, and J. Park. Classifying and comparing attribute-based and relationship-based access control. pages 59–70, 03 2017.

[17] F. Ajili and E. Contejean. Avoiding slack variables in the solving of linear diophantine equations and inequations. *Theor. Comput. Sci.*, 173(1):183–208, 1997.

[18] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126. ACM, 2014.

[19] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A novel Firewall Management Toolkit. *ACM Transactions on Computer Systems*, 22(4):381–420, 2004.

[20] M. Bartoletti and R. Zunino. A calculus of contracting processes. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010*, pages 332–341. IEEE Computer Society, 2010.

[21] G. Batra, V. Atluri, J. Vaidya, and S. Sural. Enabling the deployment of abac policies in rbac systems. In F. Kerschbaum and S. Paraboschi, editors, *Data and Applications Security and Privacy XXXII*, pages 51–68, Cham, 2018. Springer International Publishing.

[22] L. Bauer, L. F. Cranor, R. W. Reeder, M. K. Reiter, and K. Vaniea. Real life challenges in access-control management. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 899–908, New York, NY, USA, 2009. ACM.

[23] M. Y. Becker, A. Malkis, and L. Bussard. A framework for privacy preferences and data-handling policies. Technical Report MSR–TR–2009–128, Microsoft Research, September 2009.

[24] M. Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, pages 159–168. IEEE Computer Society, 2004.

[25] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic*, pages 121–135, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[26] C. Bodei, L. Ceragioli, P. Degano, R. Focardi, L. Galletta, F. L. Luccio, M. Tempesta, and L. Veronese. FWS: analyzing, maintaining and transcompiling firewalls. *J. Comput. Secur.*, 29(1):77–134, 2021.

[27] C. Bodei, P. Degano, L. Galletta, R. Focardi, M. Tempesta, and L. Veronese. Language-independent synthesis of firewall policies. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pages 92–106, 2018.

[28] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov. Towards a fully automated and optimized network security functions orchestration. In *2019 4th International Conference on Computing, Communications and Security (ICCCS), Rome, Italy, October 10-12, 2019*, pages 1–7, 2019.

[29] G. Bruns and M. Huth. Access control via belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inf. Syst. Secur.*, 14(1):9:1–9:27, June 2011.

[30] S. Calo, D. Verma, S. Chakraborty, E. Bertino, E. Lupu, and G. Cirincione. Self-generation of access control policies. In *Proceedings of the 23Nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '18, pages 39–47, New York, NY, USA, 2018. ACM.

[31] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177 – 213, 2000.

[32] 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/`.

[33] 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/`.

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

[35] L. Ceragioli, P. Degano, and L. Galletta. Are all firewall systems equally powerful? In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, PLAS'19, page 1–17. ACM, 2019.

[36] L. Ceragioli, P. Degano, and L. Galletta. Checking the Expressivity of Firewall Languages. In M. Alvim, K. Chatzikokolakis, C. Olarte, and F. Valencia, editors, *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy*, volume 11760 of *LNCS*. Springer Nature, 2019.

[37] L. Ceragioli, P. Degano, and L. Galletta. Muac: Access control language for mutual benefits. In M. Loreti and L. Spalazzi, editors, *Proceedings of the Fourth Italian Conference on Cyber Security, Ancona, Italy, February 4th to 7th, 2020*, volume 2597 of *CEUR Workshop Proceedings*, pages 119–127. CEUR-WS.org, 2020.

[38] L. Ceragioli, P. Degano, and L. Galletta. Can my firewall system enforce this policy? *Computers & Security*, 117:102683, 2022.

[39] L. Ceragioli, L. Galletta, and M. Tempesta. From firewalls to functions and back. In P. Degano and R. Zunino, editors, *Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019.*, volume 2315 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[40] 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 E. Brinksma and K. G. Larsen, editors, *14th Computer Aided Verification, Copenhagen, DK, 2002,*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[41] P. Colombo and E. Ferrari. Access control technologies for big data management systems: literature review and future trends. *Cybersecurity*, 2, 12 2019.

[42] M. Cramer, D. A. Ambrossio, and P. Van Hertum. A logic of trust for reasoning about delegation and revocation. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pages 173–184, New York, NY, USA, 2015. ACM.

[43] J. Crampton and C. Williams. On completeness in languages for attribute-based access control. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, SACMAT '16, pages 149–160, New York, NY, USA, 2016. ACM.

[44] F. Cuppens, N. Cuppens-Boulahia, J. García-Alfaro, T. Moataz, and X. Rimasson. Handling Stateful Firewall Anomalies. In *Proceedings of the 27th IFIP Information Security and Privacy Conference, SEC 2012*, pages 174–186, 2012.

[45] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A Formal Approach to Specify and Deploy a Network Security Policy. In *proc. of 2nd IFIP FAST*, pages 203–218, 2004.

[46] S. Damen, J. den Hartog, and N. Zannone. Collac: Collaborative access control. In *2014 International Conference on Collaboration Technologies and Systems, CTS 2014, Minneapolis, MN, USA, May 19-23, 2014*, pages 142–149, 2014.

[47] J. den Hartog and N. Zannone. Collaborative access decisions: Why has my decision not been enforced? In *Information Systems Security - 12th International Conference, ICISS 2016, Jaipur, India, December 16-20, 2016, Proceedings*, pages 109–130, 2016.

[48] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, may 1976.

[49] C. Diekmann. net-network: Public Collection of firewall dumps. `https://github.com/diekmann/net-network`, 2017.

[50] C. Diekmann, L. Hupel, J. Michaelis, M. P. L. Haslbeck, and G. Carle. Verified iptables firewall analysis and verification. *J. Autom. Reasoning*, 61(1-4):191–242, 2018.

[51] C. Diekmann, J. Michaelis, M. P. L. Haslbeck, and G. Carle. Verified iptables Firewall Analysis. In *Proceedings of the 15th IFIP Networking Conference, Vienna, Austria, May 17-19, 2016*, pages 252–260, 2016.

[52] E. Fehr, U. Fischbacher, and S. Gächter. Strong reciprocity, human cooperation, and the enforcement of social norms. *Human Nature*, 13(1):1–25, Mar 2002.

[53] D. Ferraiolo and D. Kuhn. Role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4, 09 1997.

[54] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. D. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15*, pages 469–483, 2015.

[55] S. N. Foley and U. Neville. A Firewall Algebra for OpenStack. In *Proceedings of the 3rd IEEE Conference on Communications and Network Security, CNS 2015*, pages 541–549, 2015.

[56] P. Fong. Relationship-based access control: Protection model and policy language. pages 191–202, 01 2011.

[57] R. Frohardt, B.-Y. E. Chang, and S. Sankaranarayanan. Access nets: Modeling access to physical spaces. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 184–198, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[58] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[59] Google. Android open source project. https://source.android.com/.

[60] P. Gordan. Ueber die Auflösung linearer Gleichungen mit reellen Coefficienten, Mar. 1873.

[61] M. G. Gouda and A. X. Liu. Structured Firewall Design. *Computer Networks*, 51(4):1106–1120, 2007.

[62] D. Grift. openwrt selinux policy. `https://git.defensec.nl/?p=selinux-policy.git;a=summary`.

[63] D. Grift. Selinux example policy cilpolicy. `https://github.com/doverride/cilpolicy`.

[64] D. Grift. Selinux example policy dspp5. `https://github.com/DefenSec/dssp5`.

[65] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134, 2005.

[66] R. Haines. The SELinux Notebook. https://github.com/SELinuxProject/selinux-notebook.

[67] W. T. Hallahan, E. Zhai, and R. Piskac. Automated repair by example for firewalls. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 220–229, Oct 2017.

[68] S. Hazelhurst. Algorithms for analysing firewall and router access lists. *CoRR*, cs.NI/0008006, 2000.

[69] S. Hazelhurst, A. Attar, and R. Sinnappan. Algorithms for improving the dependability of firewall and filter rule lists. In *2000 International Conference on Dependable Systems and Networks (DSN 2000)*, pages 576–585. IEEE Computer Society, 2000.

[70] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for selinux mls policy. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, SACMAT '07, pages 91–100, New York, NY, USA, 2007. ACM.

[71] J. A. Hoagland, R. Pandey, and K. N. Levitt. Security policy specification using a graphical approach, 1998.

[72] H. Hu, G.-J. Ahn, Z. Zhao, and D. Yang. Game theoretic analysis of multiparty access control in online social networks. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, SACMAT '14, page 93–102, New York, NY, USA, 2014. Association for Computing Machinery.

[73] V. Hu, D. Kuhn, and D. Ferraiolo. Attribute-based access control. *Computer*, 48:85–88, 02 2015.

[74] J. Hudelmaier. An o(n log n)-space decision procedure for intuitionistic propositional logic. *J. Log. Comput.*, 3(1):63–75, 1993.

[75] J. Hurd, M. Carlsson, B. Letner, and P. White. Lobster: A domain specific language for selinux policies. Technical report, Galois, Inc., 2008.

[76] B. Im, A. Chen, and D. S. Wallach. An historical analysis of the SE-Android Policy Evolution. In *Procs. 34th Annual Computer Security Applications Conference, San Juan, PR, USA, December 3-7, 2018*, pages 629–640. ACM, 2018.

[77] 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.

[78] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical report, Microsoft, 2014.

[79] A. Jeffrey and T. Samak. Model checking firewall policy configurations. In *Proceedings of the 10th IEEE International Symposium on Policies for Distributed Systems and Networks, POLICY 2009*, pages 60–67, 2009.

[80] M. I. Kanovich. Linear logic as a logic of computations. *Ann. Pure Appl. Log.*, 67(1-3):183–212, 1994.

[81] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*, pages 113–126, 2012.

[82] M. Kolár, M. C. F. Gago, and J. López. Policy languages and their suitability for trust negotiation. In F. Kerschbaum and S. Paraboschi, editors, *Data and Applications Security and Privacy XXXII - 32nd Annual IFIP WG 11.3 Conference,Proceedings*, volume 10980 of *LNCS*, pages 69–84. Springer, 2018.

[83] B. W. Lampson. Dynamic protection structures. In *Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, AFIPS '69 (Fall), pages 27–38, New York, NY, USA, 1969. ACM.

[84] C. Langenhan, M. Weber, M. Liwicki, F. Petzold, and A. Dengel. Graph-based retrieval of building information models for supporting the early design stages. *Advanced Engineering Informatics*, 27(4):413 – 426, 2013.

[85] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi. A rigorous framework for specification, analysis and enforcement of access control policies. *IEEE Transactions on Software Engineering*, 45(1):2–33, Jan 2019.

[86] R. M. Marmorstein. *Formal Analysis of Firewall Policies*. PhD thesis, College of William and Mary, May 2008.

[87] A. J. Mayer, A. Wool, and E. Ziskind. Fang: A Firewall Analysis Engine. In *proc. of the 21st IEEE S&P 2000*, pages 177–187, 2000.

[88] E. W. Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, page 238–246, New York, NY, USA, 1981. Association for Computing Machinery.

[89] P. Mehregan and P. Fong. Policy negotiation for co-owned resources in relationship-based access control. 06 2016.

[90] Microsoft Research. The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`.

[91] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[92] U. Morelli and S. Ranise. Assisted authoring, analysis and enforcement of access control policies in the cloud. In S. De Capitani di Vimercati and F. Martinelli, editors, *ICT Systems Security and Privacy Protection*, pages 296–309, Cham, 2017. Springer International Publishing.

[93] M. Müller-Olm, D. A. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In A. Cortesi and G. Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *LNCS*, pages 330–354. Springer, 1999.

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

[95] M. Narouei, H. Takabi, and R. Nielsen. Automatic extraction of access control policies from natural language documents. *IEEE Transactions on Dependable and Secure Computing*, PP:1–1, 03 2018.

[96] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of the 24th Large Installation System Administration Conference, LISA 2010*, 2010.

[97] H. Nergaard, N. Ulltveit-Moe, and T. Gjosaeter. A scratch-based graphical policy editor for xacml. In *2015 International Conference on Information Systems Security and Privacy (ICISSP)*, pages 1–9, Feb 2015.

[98] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[99] B. O'Sullivan. Anomaly analysis for physical access control security configuration. In *Proceedings of the 2012 7th International Conference on Risks and Security of Internet and Systems (CRiSIS)*, CRISIS '12, pages 1–8, Washington, DC, USA, 2012. IEEE Computer Society.

[100] F. Paci, A. C. Squicciarini, and N. Zannone. Survey on access control for community-centered collaborative systems. *ACM Comput. Surv.*, 51(1):6:1–6:38, 2018.

[101] S. M. Perez, J. Cabot, J. García-Alfaro, F. Cuppens, and N. Cuppens-Boulahia. A Model-Driven Approach for the Extraction of Network Access-Control Policies. In *Proceedings of the Workshop on Model-Driven Security Workshop, MDsec 2012*, 2012.

[102] J. Qiu, Z. Tian, C. Du, Q. Zuo, S. Su, and B. Fang. A survey on access control in the age of internet of things. *IEEE Internet of Things Journal*, 7(6):4682–4696, 2020.

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

[104] S. Rajtmajer, A. Squicciarini, C. Griffin, S. Karumanchi, and A. Tyagi. Constrained social-energy minimization for multi-party sharing in online social networks. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, page 680–688, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[105] D. Ranathunga, M. Roughan, P. Kernick, and N. Falkner. Malachite: Firewall policy comparison. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 310–317, June 2016.

[106] P. Rao, G. Ghinita, E. Bertino, and J. Lobo. Visualization for access control policy analysis results using multi-level grids. In *Proceedings of the 10th IEEE International Conference on Policies for Distributed Systems and Networks*, POLICY'09, pages 25–28, Piscataway, NJ, USA, 2009. IEEE Press.

[107] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. Fine-grained integration of access control policies. *Comput. Secur.*, 30(2-3):91–107, Mar. 2011.

[108] E. Reshetova, F. Bonazzi, and N. Asokan. Selint: An seandroid policy analysis tool. In P. Mori, S. Furnell, and O. Camp, editors, *Procs 3rd International Conference on Information Systems Security and Privacy, Porto, PT, 2017*, pages 47–58. SciTePress, 2017.

[109] R. Russell. Linux 2.4 Packet Filtering HOWTO. `http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html`, 2002.

[110] P. Samarati and S. Vimercati. Access control: Policies, models, and mechanisms. volume 2171, pages 137–196, 09 2000.

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

[112] D. Servos and S. L. Osborn. Current research and open problems in attribute-based access control. *ACM Comput. Surv.*, 49(4), jan 2017.

[113] N. Skandhakumar, F. Salim, J. Reid, and E. Dawson. Physical access control administration using building information models. In Y. Xiang, J. Lopez, C.-C. J. Kuo, and W. Zhou, editors, *Cyberspace Safety and Security*, pages 236–250, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

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

[115] W. Stallings and L. Brown. *Computer Security: Principles and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2014.

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

[117] J. M. Such and M. Rovatsos. Privacy policy negotiation in social media. *ACM Trans. Auton. Adapt. Syst.*, 11(1), feb 2016.

[118] G. Suntaxi, A. A. El Ghazi, and K. Böhm. Mutual authorizations: Semantics and integration issues. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, SACMAT '19, pages 213–218, New York, NY, USA, 2019. ACM.

[119] The Netfilter Project. Traversing of Tables and Chains. `http://www.iptables.info/en/structure-of-iptables.html`.

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

[121] FireWall Synthesizer (FWS): Tool and Examples. `https://github.com/secgroup/fws`.

[122] P. Tsankov, M. T. Dashti, and D. A. Basin. Access control synthesis for physical spaces. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 443–457, 2016.

[123] C. Tsigkanos, L. Pasquale, C. Ghezzi, and B. Nuseibeh. On the interplay between cyber and physical spaces for adaptive security. *IEEE Trans. Dependable Sec. Comput.*, 15(3):466–480, 2018.

[124] C. Tsigkanos, L. Pasquale, C. Menghi, C. Ghezzi, and B. Nuseibeh. Engineering topology aware adaptive security: Preventing requirements violations at runtime. In *IEEE 22nd International Requirements Engineering Conference, RE 2014, Karlskrona, Sweden, August 25-29, 2014*, pages 203–212, 2014.

[125] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone. Analysis of xacml policies with smt. In R. Focardi and A. Myers, editors, *Principles of Security and Trust*, pages 115–134, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[126] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone. Formal analysis of xacml policies using smt. *Comput. Secur.*, 66(C):185–203, may 2017.

[127] F. Valenza, S. Spinoso, and R. Sisto. Formally specifying and checking policies and anomalies in service function chaining. *J. Network and Computer Applications*, 146, 2019.

[128] Q. Xiao and K.-L. Tan. Peer-aware collaborative access control in social networks. In *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 30–39, 2012.

[129] L. Xu, N. Shah, L. Chen, N. Diallo, Z. Gao, Y. Lu, and W. Shi. Enabling the sharing economy: Privacy respecting contract based on public blockchain. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, BCC '17, page 15–21, New York, NY, USA, 2017. Association for Computing Machinery.

[130] 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.*, 92-D, 11:2196–2206, 2009.

[131] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *Proceedings of the 27th IEEE Symposium on Security and Privacy, S&P 2006*, pages 199–213, 2006.

Appendix A

Technical Details and Proofs of Chapter 2

A.1 IFCL Normal Form: Proofs

The following property is used in the proof of Lemma A.1 below. Intuitively, it says that, if given a ruleset R a packet p matches no control flow rule, then the packet matches a rule in the unfolded ruleset $(\!(R)\!)_I^{true} \bullet$ with the same target.

Property A.1. Given a ruleset R, if $p \Vdash_R (t, i)$ with $t \in \{\texttt{ACCEPT}, \texttt{DROP}, \texttt{NAT}(n_1, n_2), \texttt{CHECK-STATE}(X), \texttt{MARK}\}$ then there exist m and I such that $p \models_{(\!(R)\!)_I^{true} m} (t, j)$ for some j.

Proof. We know that $R = [(\phi_1, t_1), \ldots, (\phi_n, t_n)]$ and that $\phi_k(p)$ does not hold for $k < i$. Furthermore, we know that our unfolding algorithm replaces each rule $r_k = (\phi_k, t_k)$ with a list of rules whose predicates have a suitable ϕ'_k as a conjunct for I not containing R' and m the tag of p (so p satisfies ϕ_k iff it satisfies ϕ'_k). Otherwise r_k is replaced with $(\phi'_k \wedge \mathit{true}, \texttt{DROP})$. Since $\phi'_k(p)$ is false for $k < i$, the new rules do not apply, whereas the rule r_i still applies. Moreover, consider the case when a rule $r_k = (\phi_k, \texttt{RETURN})$ occurs in the ruleset for $k < i$. In the resulting ruleset this rule is canceled and the rules that follow it are rewritten by adding the conjunct $\neg\phi'_k$. Since r_k does not apply, we know that $\neg\phi_k(p)$ is *true*, and this does not affect the evaluation of the other rules, so that the rule r_i still matches. □

The following lemma shows the correctness of our unfolding procedure. Intuitively, it tells us that packet p is evaluated in the same way by a ruleset R and by its unfolding $(\!(R)\!)$.

Lemma A.1. Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and given a ruleset R such that $c(q) = R$ for some node q of $\mathcal{C}$, we have that for $t \in \{\texttt{ACCEPT}, \texttt{DROP}\}$

$$\forall p, s.p, s \models^{\epsilon}_R (t, p') \iff p, s \models^{\epsilon}_{(\!(R)\!)} (t, p')$$

Proof. We prove the following stronger statement from which the lemma follows as a particular case:

$$\forall p, s, S.\ p, s \models^{S}_{R} (t, p') \iff p, s \models^{\epsilon}_{(\!|\, R \,|\!)^{I}_{true} m} (t, p')$$

with $t \in \{\texttt{ACCEPT}, \texttt{DROP}\}$ and $I = \mathit{flatten}(S) \cup \{R\}$ for some m, where the function *flatten* is recursively defined as follows, where $X \in \{R, \overline{R}, \overline{R_i}, R_{i+1}\}$:

$$\mathit{flatten}(\epsilon) = \emptyset \qquad \mathit{flatten}(X \cdot S) = \mathit{flatten}(S) \cup \{X\}$$

First we prove the "if" case by induction on the derivations of $p, s \models^{S}_{R}$, and then by cases on the last rule applied.

- **rule (1)**. By the premise of the rule, it holds that $p \Vdash_R (t, i)$ for some i, where $t \in \{\texttt{ACCEPT}, \texttt{DROP}\}$. The thesis follows by applying Property A.1. Note that Property A.1 holds for all sets I and in particular for $I = \mathit{flatten}(S) \cup \{R\}$.

- **rules (2) and (4)**. Similar to the rule (1).

- **rule (3)**. By the premise of the rule, we have that $p \Vdash_R (\texttt{CHECK-STATE}(X), i)$, $p \not\Vdash_s$ and $p, s \models^{S}_{R_{i+1}} (t, p')$. Moreover, by Property A.1, we have that $p \Vdash_{(\!|\, R \,|\!)^{true}_{I}} (\texttt{CHECK-STATE}(X), j)$ holds for some j and, by induction hypothesis, that $p, s \models^{S}_{(\!|\, R_{i+1} \,|\!)^{I}_{true} m} (t, p')$ for $I = \mathit{flatten}(S) \cup \{R\}$. Thus, the thesis holds by applying the rule (3).

- **rule (5)**. By the premise of the rule, it holds $p \Vdash_R (\texttt{GOTO(R')}, i)$, i.e., the rule $r_i = (\phi_i, \texttt{GOTO(R')})$ matches, $R' \notin S$ and $p, s \models^{\overline{R} \cdot S}_{R'} (t, p')$. By induction hypothesis we have that $p, s \models^{\overline{R} \cdot S}_{(\!|\, R' \,|\!)^{I'}_{true} m} (t, p')$ with $I' = \mathit{flatten}(\overline{R} \cdot S) \cup \{R'\}$. From the definition of the unfolding procedure we know that for all I not including R' the ruleset $(\!|\, R \,|\!)^{I}_{true} m$ includes all the rules of $(\!|\, R' \,|\!)^{I \cup \{R'\}}_{true} m$ prefixed by the predicate ϕ_i as a conjunct. Since ϕ_i is true for p, there is not change in the validity of the rule matching p in $(\!|\, R' \,|\!)^{I'}_{true} m$ that continues to match p also in $(\!|\, R \,|\!)^{I}_{true} m$, for some j and for all I not including R'. So can conclude the thesis $p, s \models^{S}_{(\!|\, R \,|\!)^{I}_{true} m} (t, p')$ by taking $I = \mathit{flatten}(S) \cup \{R\}$.

- **rule (7)**. Similar to rule (5).

- **rule (6)**. By the premise of the rule, we know that $p \Vdash_R (\texttt{GOTO(R')}, i)$, i.e., there exists $\phi_i(p)$ that holds, and $R' \in S$. Hence, taking $I = \mathit{flatten}(S) \cup \{R\}$ we have that $R' \in I$, and in $(\!|\, R \,|\!)^{I}_{true} m$ the rule matching r_i is replaced with $(\mathit{true} \wedge \phi, \texttt{DROP})$ and we can obtain the thesis.

- **rule (8)**. Similar to the case of rule (6).

- **rule (9)**. By the premise of the rule, we know that $p \Vdash_R (\textsc{return}, i)$, $pop^*(S) = (R', S')$ and $p, s \models^{S'}_{R'} (t, p')$. In the ruleset $\llparenthesis R \rrparenthesis$ the rule r_i is skipped and all the rules after the r_i are rewritten by adding the negation of the predicate ϕ_i of r_i as a conjunct; in these way none of these new rules match p. Thus, $p \not\models_{\llparenthesis R \rrparenthesis_{true} Im}$ for any I. The thesis follows by applying the induction hypothesis and then, the rule (11).

- **rule (10)**. It is essentially as the proof of rule (9) except that at the end the thesis follows applying the rule (12).

- **rule (11)**. By the premise of the rule, we know $p \nVdash_R$, $S \neq \epsilon$, $pop^*(S) = (R', S')$ and $p, s \models^{S'}_{R'} (t, p')$. Trivially it holds that if $p \nVdash_R$ then $p \not\models^I_{\llparenthesis R \rrparenthesis_{true} m}$ for all I. Thus, the thesis follows by applying the induction hypothesis and the rule (11).

- **rule (12)**. By the premise of the rule, it holds that $p \nVdash_R$ and $S = \epsilon \vee pop^*(S) = \epsilon$. Since if $p \nVdash_R$ then $p \not\models^I_{\llparenthesis R \rrparenthesis_{true} m}$ for all I, the thesis trivially holds by applying rule (12).

- **rule (13)**. The premises of the rule guarantee that p matches the i^{th} rule with target $\textsc{mark}(m)$ and that the execution of the ruleset proceeds with $p[tag \mapsto m]$. By Property A.1 p matches in the unfolded ruleset the j^{th} rule with target $\textsc{mark}(m)$ for some j. Now the induction hypothesis applies yielding $p[tag \mapsto m], s \models^{\epsilon}_{\llparenthesis R_{j+1} \rrparenthesis^I_{true} m} (t, p')$, so fulfilling the premises of the rule (13).

To prove the case "only if" we proceed by contradiction by assuming that $p, s \models^{\epsilon}_{\llparenthesis R \rrparenthesis^I_{true} m} (t, p')$ for $I = \mathit{flatten}(S) \cup \{R\}$, but that $p, s \models^{\epsilon}_R (t_1, p'_2)$ with $(t, p') \neq (t_1, p'_2)$. By applying the just proved case "if" of the Lemma we know that $p, s \models^{\epsilon}_{\llparenthesis R \rrparenthesis^{I'}_{true} m} (t_1, p'_2)$ for $I' = \mathit{flatten}(S) \cup \{R\}$. Contradiction. □

The following lemma guarantees that the evaluations in the slave transition system for a firewall and its unfolded version are the same.

Lemma A.2. Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall and $\llparenthesis \mathcal{F} \rrparenthesis$ its unfolding. Let $(q, s, p) \rightarrow_X (q', s, p')$ be a step of the slave transition system performed by the firewall $X \in \{\mathcal{F}, \llparenthesis \mathcal{F} \rrparenthesis\}$. Given a node q of $\mathcal{C}$ we have that

$$(q, s, p) \rightarrow_{\mathcal{F}} (q', s, p') \iff (q, s, p) \rightarrow_{\llparenthesis \mathcal{F} \rrparenthesis} (q', s, p').$$

Proof. Assume $(q, s, p) \rightarrow_{\mathcal{F}} (q', s, p')$. By definition of the slave transition system we know that $c(q) = R$, $p, s \models^{\epsilon}_R (\textsc{accept}, p')$ and $\delta(q, p') = q'$. By Definition 2.9 and by Lemma A.1, we have $c'(q) = \llparenthesis R \rrparenthesis$ and $p, s \models^{\epsilon}_{\llparenthesis R \rrparenthesis} (\textsc{accept}, p')$, so we can prove $(q, s, p) \rightarrow_{\llparenthesis \mathcal{F} \rrparenthesis} (q', s, p')$. The case "only if" follows the same schema. □

Now we can easily prove Theorem 2.1.

Theorem 2.1 (Correctness of unfolding)**.** Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall and $(\!|\mathcal{F}|\!)$ its unfolding. Let $s \xrightarrow{p,p'}_X s'$ be a step of the master transition system performed by the firewall $X \in \{\mathcal{F}, (\!|\mathcal{F}|\!)\}$. Then

$$s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{(\!|\mathcal{F}|\!)} s'.$$

Proof. Assume $s \xrightarrow{p,p'}_{\mathcal{F}} s'$. By the premise of the rule, we know that there exists a sequence of steps of the slave transition system such that $(q_i, s, p) \rightarrow^+_{\mathcal{F}} (q_f, s, p')$. By repeatedly using Lemma A.2 we have that $(q_i, s, p) \rightarrow^+_{(\!|\mathcal{F}|\!)} (q_f, s, p')$. Thus, the thesis follows by applying the rule of the master transition system.
The "only if" case follows the same schema. □

A.2 Decompilation: Proofs

The following property is an immediate consequence of Definition 2.3.

Property A.2. Given a ruleset R and a rule $r' = (\phi', t')$, if $p \Vdash_R (t, i)$ and $\phi'(p)$ does not hold then $p \models_{r';R} (t, i)$.

The following lemma guarantees the correctness of the predicate definition in Table 2.2.

Lemma 2.1. Given a ruleset R we have that

1. $\forall p, s.\ p, s \models^{\epsilon}_R (\textsc{accept}, p') \implies P_R(p, p')$; and
2. $\forall p, p'.\ P_R(p, p') \implies \exists s.\ p, s \models^{\epsilon}_R (\textsc{accept}, p')$

Proof. **(1)**. By induction on the depth of the derivation of $p, s \models^{\epsilon}_R (\textsc{accept}, p')$, and by cases on the last rule applied. Note that since we have considered unfolded firewalls the only rules we consider are (1), (2), (3), (4), (12) and (13).

- **rule (1)**. By the premise of the rule, we know that $p \Vdash_R (\textsc{accept}, i)$ (i.e., there exists $r_i = (\phi_i, \textsc{accept})$ and $\phi_(p)$ holds) and $p = p'$. By Table 2.2, we know that the formula P_R has a disjunct $\phi_i(p) \wedge (p = p')$ that holds, proving the thesis.
- **rule (2)**. By the premise of the rule $p \Vdash_R (\textsc{check-state}(X), i)$ (i.e., $\phi_i(p)$ holds), $p \vdash_s \alpha$ and $p' = \mathit{establ}(\alpha, X, p)$ (p is rewritten as p'). The thesis follows because, by Table 2.2, P_R contains a disjunct $\phi(p) \wedge p' \in tr(p, *{:}*, *{:}*, X)$ that holds.
- **rules (3) and (13)**. Trivial using the induction hypothesis.
- **rule (4)**. By the rule, we have $p \Vdash_R (\textsc{nat}(n_1, n_2), i)$ (i.e., $\phi_i(p)$ holds) and $p' = \mathit{nat}(p, s, d_s, n_s)$. By Table 2.2, we know that there is a disjunct $\phi(p) \wedge p' \in tr(p, d_s, n_s, \leftrightarrow)$ that holds, thus proving the thesis.

- **rule (12)**. This case applies if the default policy of the ruleset is ACCEPT and $p' = p$. By Table 2.2, the disjunct $dp(R) \wedge p' = p$ holds, proving the thesis.

(2). By induction on the length of the ruleset R, and then by cases on its first rule. When R is empty, the formula P_R is $dp(R) \wedge p = \tilde{p}$ and $p \nvdash_R$. Thus, the thesis follows for each state s by using the rule (12) with an empty stack. If R is not empty we consider all the possible different cases. Note that apart from the case for DROP our translation in Table 2.2 procedure creates mutually exclusive disjunctions.

- **case** (ϕ, ACCEPT). If the first disjunct holds, the thesis follows by applying the rule (1) in any state s. If the second one holds, the thesis follows by induction hypothesis and by Property A.2.
- **case** (ϕ, DROP). Just as the second case above.
- **case** $(\phi, \text{NAT}(d_s, n_s))$. If the first disjunct holds we know that $p \Vdash_R (\text{NAT}(n_1, n_2), i)$ and that p' is one of the possible translation of p. To prove the thesis it suffices to apply the rule (4), taking a state s such that $p' = nat(p, s, n_d, n_s)$. If the second disjunct holds, the thesis follows by induction hypothesis and by applying the Property A.2.
- **case** $(\phi, \text{CHECK-STATE}(X))$. If the first disjunct holds, we have that $\phi(p)$ is true and p' is obtained by rewriting p. Thus, the thesis follows by applying the rule (2) in a state any state s where $p \vdash_s \alpha$ and $p' = establ(\alpha, X, p)$. If the second disjunct holds, the thesis follows by induction hypothesis and by Property A.2.
- **case** $(\phi, \text{MARK}(m))$. By induction hypothesis, similarly to the ACCEPT case.

□

The following auxiliary lemma establishes the correspondence between the executions in the slave transition system from a node q and the formula built for the same node q.

Lemma A.3. Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and a node q of $\mathcal{C}$, we have that for some I

1. $\forall s, p.\ (q, s, p) \rightarrow^* (q_f, s, \tilde{p}) \implies \mathcal{P}_q^I(p, \tilde{p})$
2. $\forall q, p, \tilde{p}.\ \mathcal{P}_q^I(p, \tilde{p}) \implies \exists s.(q, s, p) \rightarrow^* (q_f, s, \tilde{p})$
3. $\forall s, p.\ (q, s, p) \not\rightarrow^* (q_f, s, \tilde{p}) \Rightarrow \mathcal{D}_q^I(p) = true$

Proof. **(1)**. By induction on the length of the derivation of $(q, s, p) \rightarrow^n (q_f, s, \tilde{p})$. In the case of $n = 0$ the thesis trivially holds. We assume that the statement holds for derivation of length n and we prove the case $n + 1$. Thus, there is a derivation $(q, s, p) \rightarrow (q', s, p') \rightarrow^n (q_f, s, \tilde{p})$. By the premise of the slave

transition system, we know that $p, s \models^{\epsilon}_{c(q)} (\textsc{accept}, p')$ and $\delta(q, p') = q'$. By Lemma 2.1 (1), we know that $P_{c(q)}(p, p')$ is true and by Definition 2.5 that $\psi(p')$ holds. By applying the induction hypothesis we have $\mathcal{P}^I_{q'}(p, \tilde{p})$ and the thesis follows.
(2). Note that since $\mathcal{P}^I_q(p, \tilde{p})$ holds there exist n packets p_i and a path of length n in the control diagram of the firewall $q_1 \xrightarrow{\psi_1(p_1)} q_2 \xrightarrow{\psi(p_2)} \ldots \xrightarrow{\psi(p_n)} q_f$ such that $q_1 = q$, $p_1 = p$, $p_n = \tilde{p}$ and $\wedge_{i=1}^n \psi_i(p_i)$ holds. We proceed by induction on the length n of this path. For $n = 0$ the thesis trivially holds. We assume the statement valid for n and consider given a path of length $n + 1$: $q \xrightarrow{\psi(p)} q_1 \xrightarrow{\psi_1(p_1)} q_2 \xrightarrow{\psi(p_2)} \ldots \xrightarrow{\psi(p_n)} q_f$ with $p_n = \tilde{p}$. Since $\mathcal{P}^I_q(p, \tilde{p})$ holds, we know that $P_{c(q)}(p, p')$ holds for some p'. By applying Lemma 2.1 (2), we know that there exists a state s such that $p, s \models^{\epsilon}_R (\textsc{accept}, p')$. Since in the path we have the arc $q \xrightarrow{\psi(p)} q_1$ we also know that $\delta(q, p') = q_1$. The thesis follows by taking $p' = p_1$ and by induction hypothesis.
(3) Similar to the proof of case **(1)**. □

Now we can prove the following theorem, which states the correspondence between the logical formulation and the operational semantics.

Theorem 2.2 (Correctness of the logical characterization)**.** Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and its corresponding predicate $\mathcal{P}_\mathcal{F}$, for all packets p we have that $\mathcal{P}_\mathcal{F}(p, p') \vee \mathcal{D}_\mathcal{F}(p)$ holds and

1. $s \xrightarrow{p,p'} s \uplus (p, p') \implies \mathcal{P}_\mathcal{F}(p, p')$
2. $\nexists p', s'.\ s \xrightarrow{p,p'} s' \implies \mathcal{D}_\mathcal{F}(p)$
3. $\mathcal{P}_\mathcal{F}(p, p') \implies \exists s. s \xrightarrow{p,p'} s \uplus (p, p')$

Proof. **(1)** and **(2)** The thesis follows by taking the premise of the master transition system and by applying Lemma A.3 (1) and (3), respectively
(3). The thesis follows by applying first the Lemma A.3 (2), and then the rule of the master transition system, using the same state s given by Lemma A.3 (2). □

A.3 Compilation: Proofs

Theorem 2.3. Let $\mathcal{F}_C$ be a *compiled firewall* of τ and let p be a packet, then

$$\tau(p) \neq \lambda_\perp \Leftrightarrow \exists p'. \mathcal{P}_{\mathcal{F}_C}(p, p').$$

Proof. ($\Rightarrow$) We show that a packet p accepted by $\mathcal{F}_S$ never matches a $\textsc{drop}$ rule in the rulesets of $\mathcal{F}_C$. Recall that a configuration cannot accept and drop the same packet. Since rulesets have a default $\textsc{accept}$ policy, we consider only R_{fil} and $R_{dnat} \mathbin{@} R_{fil}$, $R_{snat} \mathbin{@} R_{fil}$ which are the only rulesets containing the rule $(true, \textsc{drop})$ at the end. We distinguish two cases:

- p is accepted by rule $r = (\phi, \texttt{NAT}(n_d, n_s)) \in R_S$ in $\mathcal{F}_S$. By construction of the rulesets, in $\mathcal{F}_C$ the packet will be tagged with some mark m in the first non-empty ruleset in the path $\pi_{\mathcal{F}_C}(p)$; then:
  - if processed by R_{fil}, it is accepted by the rule $(tag(p) \neq \bullet, \texttt{ACCEPT})$;
  - if processed by $R_{dnat} \mathbin{@} R_{fil}$, it is accepted (and translated) according to the NAT rule $(tag(p) = m, \texttt{NAT}(n_d, \star))$ in R_{dnat};
  - similarly, if processed by $R_{snat} \mathbin{@} R_{fil}$, it is accepted by rule $(tag(p) = m, \texttt{NAT}(\star, n_s))$ in R_{snat}.
- p is accepted by rule $r = (\phi, \texttt{ACCEPT}) \in R_S$ in $\mathcal{F}_S$, hence it is not subject to NAT. Algorithm 2 places the rule r in R_{fil}, therefore the packet is accepted when rulesets R_{fil}, $R_{dnat} \mathbin{@} R_{fil}$ are $R_{snat} \mathbin{@} R_{fil}$ are traversed.

($\Leftarrow$) Since $\mathcal{P}_{\mathcal{F}_C}(p, p'')$, there exist a path $\pi_{\mathcal{F}_C}(p) = \langle q_1, \ldots, q_n \rangle$ and $n+1$ packets $p_1, \ldots, p_{n+1}$ such that $P_{c(q_j)}(p_j, p_{j+1})$ for $j \in [1..n]$ where $p_1 = p$ and $p_{n+1} = p''$. By definition of compiled firewall, there exists $j \in [1..n]$ such that $c(q_j) \in \{R_{fil}, R_{dnat} \mathbin{@} R_{fil}, R_{snat} \mathbin{@} R_{fil}\}$. Assume $c(q_j) = R_{fil}$, we distinguish two cases:

- $tag(p_{j+1}) \neq \bullet$: by Algorithm 2, there exists a MARK rule in the first non-empty ruleset of $\pi_{\mathcal{F}_C}(p)$ such that $\phi(p)$ holds. Hence, there exists some rule $r = (\phi, \texttt{NAT}(n_d, n_s)) \in R_S$ that accepts p translated as p'.
- $tag(p_{j+1}) = \bullet$: packet p_{j+1} is accepted by some rule $r = (\phi, \texttt{ACCEPT}) \in R_{fil}$; we also have $p_{j+1} = p$ since the packet is not tagged, thus it is not transformed by any NAT in $\mathcal{F}_C$. Since rule $r \in R_S$ by construction, $\mathcal{F}_S$ accepts p without translation.

The cases $c(q_j) \in \{R_{dnat} \mathbin{@} R_{fil}, R_{snat} \mathbin{@} R_{fil}\}$ follow a similar scheme. □

Property A.3. Let p be a packet accepted by a compiled firewall $\mathcal{F}_C$. We have that:

1. if p is accepted without NATs it is never tagged by the firewall;
2. if p accepted with NATs, it is tagged exactly once in the first non-empty ruleset of $\pi_{\mathcal{F}_C}(p)$.

Proof. **(1)** If a packet is not tagged in the first non-empty ruleset of the path $\pi_{\mathcal{F}_C}(p)$, all the conditions ϕ in the MARK rules do not apply. Thus, none of the (possible) NAT rules applies and the packet is left unchanged. Therefore, subsequent evaluations of the marking rules still do not apply.
(2) Straightforward, MARK rules include the check $tag(p) = \bullet$ in their conditions. Marking occurs in the first node of $\pi_{\mathcal{F}_C}(p)$ that contains a non-empty ruleset, i.e., a ruleset different from $R_\epsilon(p)$. □

Theorem 2.4. Let p be a packet accepted by both $\mathcal{F}_S$ and $\mathcal{F}_C$; let $\beta = tc(\pi_{\mathcal{F}_C}(p))$; and let $p'' \approx t_\beta(p, p')$ for some p'. We have that

$$\mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \mathcal{P}_{\mathcal{F}_C}(p, p'')$$

with $p' = p''$ when $\beta = nat$ or $p = p'$.

Proof. ($\Rightarrow$) Assume $\mathcal{P}_{\mathcal{F}_S}(p, p')$. Let $\pi_{\mathcal{F}_C}(p) = \langle q_1, \ldots, q_n \rangle$ with $q_1 = q_i$ and $q_n = q_f$ and we consider $n + 1$ intermediate packets $p_1, \ldots, p_{n+1}$ with $p_1 = p$ and $p_{n+1} = p''$. We proceed by cases on the translation $tt(p, p')$.

- If $tt(p, p') = id$, by Property A.3 we know that p is never tagged by $\mathcal{F}_C$. Since NAT rules are applied only to tagged packets, all rulesets in $\pi_{\mathcal{F}_C}(p)$ accept the packet without translations, i.e., $\mathcal{P}_{\mathcal{F}_C}(p, p)$ holds. We have $t_\beta(p, p') = p' = p$ for all possible values of β, therefore the thesis hold.
- If $tt(p, p') = nat$, by Property A.3, we know that p is tagged in the first non-empty ruleset $c(q_j)$ by rule $(\phi \wedge tag(p) = \bullet, \texttt{MARK}(m))$, i.e., $tag(p_{j+1}) = m$.
  - If $\beta = id$, we have $p_{j+1} = \cdots = p_{n+1} = p''$ since the packet traverses and is accepted by rulesets that do not contain NAT rules. Therefore $p'' \approx t_{id}(p, p')$ and the thesis hold.
  - If $\beta = nat$, let $k, l \in [j + 1..n]$ the smallest indexes such that $dnat \in \gamma(c(q_k))$ and $snat \in \gamma(c(q_l))$. Without loss of generality, we assume $k < l$ (the other case is analogous). We have that $p_{j+1} = \cdots = p_k$. Packet p_k is processed by ruleset $c(q_k)$ that applies the DNAT translation associated with tag m, i.e., $p_{k+1} \approx p[da \mapsto da(p')]$. Then we have $p_{k+1} = \cdots = p_l$. Packet p_l is processed by ruleset $c(q_l)$ that applies the DNAT translation associated with tag m, thus we have $p_{l+1} \approx p'$. Finally we have $p_{l+1} = \cdots = p_{n+1} \approx p'$. Since $t_{nat}(p, p') = p'$, the thesis hold.
  - For $\beta \in \{dnat, snat\}$, the proof is a simplified version of $\beta = nat$.
- Proofs for cases $tt(p, p') = snat$ and $tt(p, p') = dnat$ are similar to the case $tt(p, p') = nat$.

($\Leftarrow$) Assume $\mathcal{P}_{\mathcal{F}_C}(p, p'')$. We distinguish two cases, depending on the fact that p'' is tagged or not.

- If $tag(p'') = \bullet$, we know that p'' has been accepted without NATs, i.e., $p = p''$. By definition of $\mathcal{F}_C$, the path $\pi_{\mathcal{F}_C}(p)$ has a node associated with a ruleset R in $\{R_{fil}, R_{dnat} \mathbin{@} R_{fil}, R_{snat} \mathbin{@} R_{fil}\}$. Since p is accepted by R, it means that p is accepted by one of the filtering rules taken from R_S. Therefore we have $\mathcal{P}_{\mathcal{F}_S}(p, p)$, $p'' \approx t_\beta(p, p)$ for any β and the thesis hold.
- Let $tag(p'') = m$. By Property A.3 we know that the packet is tagged only once during its processing and tagging occurs inside the first non-empty ruleset of $\pi_{\mathcal{F}_C}(p)$. By Algorithm 2, we know that there exists a

rule $r = (\phi, \texttt{NAT}(n_d, n_s)) \in R_S$ for some n_d, n_s that accepts p as p', i.e. $\mathcal{P}_{\mathcal{F}_S}(p, p')$. Moreover, the same ranges are used in the NAT rules that have translated p into p'' during the traversal of the path $\pi_{\mathcal{F}_C}(p)$. Hence we have $p'' \approx t_\beta(p, p')$ for any β and the thesis follows.

□

A.4 Computing the Representative Pairs

We now show how to efficiently build a pair (p, t), representative of a whole $\omega_{X_1,Y,X_2} \neq \emptyset$ without actually computing this equivalence class. One can thus effectively check whether ω_{X_1,Y,X_2} is expressible by applying Theorem 2.6 to the pair (p, t). The algorithm relies on two assumptions that hold in all the firewall languages we have considered. The first requires that each predicate on the arcs is expressible as a predicate on a single field of the packet.[1] The second assumption says that given a satisfiable predicate one can mechanically build a packet satisfying it.

Given X_1, Y, X_2, the Algorithm 5 computes such a representative or fails if there are none, i.e., if $\omega_{X_1,Y,X_2} = \emptyset$. In it, we let w range over the fields of a packet and of a transformation, including $\{dIP, dPort, sIP, sPort\}$. Recall that we denote the fields of a packet p by p_w and those of a transformation t by t_w. In the same way, we split a set of predicates X in the components on the fields, typically $X = X_{dIP} \wedge X_{dPort} \wedge X_{sIP} \wedge X_{sPort}$, where $\wedge$ operates homomorphically. For each X_w, the function $\texttt{TAKE\_ONE}(X_w)$ returns an address whatsoever if $X_w = \emptyset$, or an address satisfying all the predicates in X_w (second assumption above). Note that X_w might be unsatisfiable, so making $\texttt{TAKE\_ONE}(X_w)$ and the whole algorithm fail.

The Algorithm 5 scans the fields w of the packet headers and generates an address satisfying the w component of the predicates in X_1. Then it takes a transformation t' in Y, if any, that changes the field w. In such a case, an address a is taken that satisfies X_{2_w}, and the w component of the output transformation t is set to λ_a. If no $t' \in Y$ changes the field w, the transformation t on w is id; also, the algorithm fails when the predicates X_1 and X_2 differ on w.

We prove now the correctness of Algorithm 5 . For ease of notations we call $W_\mathcal{T} \subset W$ the fields of a transformation, namely $dIP, dPort, sIP, sPort$.

Lemma A.4. For each set of transformations $Y \neq \varepsilon(\texttt{DROP})$ in $\mathbb{T}$ let $\widetilde{W_\mathcal{T}}$ be the set of fields such that $\forall w \in \widetilde{W_\mathcal{T}}. \forall t \in Y. t_w = id$, then the following holds: for all the tuples of field values $a_w \in D_w$ for $w \in W_\mathcal{T} \setminus \widetilde{W_\mathcal{T}}$, the transformation defined as λ_{a_w} for $w \in W_\mathcal{T} \setminus \widetilde{W_\mathcal{T}}$ and as id for $w \in \widetilde{W_\mathcal{T}}$ is in Y.

Proof. The thesis can be trivially verified case-by-case by definition on the elements of $\mathbb{T}$. □

[1] Since one can add new nodes without loosing expressive power, this is the same of asking each predicate on the arcs to be expressible as conjunction of one predicate for each field, possibly *true*.

Algorithm 5 Build a pair $(p, t) \in \omega_{X_1,Y,X_2}$, if any.

```
1: if Y = {λ⊥} then t = λ⊥
2: for all w ∈ W do
3:     p_w ← TAKE_ONE(X_{1_w})
4:     if Y ≠ {λ⊥} ∧ w ∈ {dIP, dPort, sIP, sPort} then
5:         if ∃t' ∈ Y.t'_w ≠ id then
6:             a ← TAKE_ONE(X_{2_w})
7:             t_w ← λ_a
8:         else if X_{1_w} ≠ X_{2_w} then FAIL
9:         else t_w ← id
10: return (p, t)
```

The correctness of Algorithm 5 is guaranteed by the following theorem.

Theorem A.1. Given a triple X_1, Y, X_2, the Algorithm 5 either returns a pair $(p, t) \in \omega_{X_1,Y,X_2} \neq \emptyset$ or it fails.

Proof. Assume that $Y = \{\lambda_\perp\}$, then by definition $(p, t) \in \omega_{X_1,Y,X_2}$ only if $t = \lambda_\perp$ and $g(p) = X_1$. The first condition is guarantee by construction of the algorithm, the second one is verified only if $\forall w \in W. \exists a$ such that the set of predicates verified by a is X_{1_w}, and this is what is returned by `TAKE_ONE`, which in turn fails if there is no such an address for some w. Hence if the condition cannot be verified the algorithm fails, otherwise it returns a pair in ω_{X_1,Y,X_2}.

If $Y \neq \{\lambda_\perp\}$ then if $\nexists p. g(p) = X_1$ then $\omega_{X_1,Y,X_2} = \emptyset$ and the algorithm fails because it implies that $\exists w \in W. \nexists a$ such that the set of predicates verified by a is X_{1_w} and hence, as we stated before, `TAKE_ONE` fails. Vice versa if such a packet exists then by definition `TAKE_ONE` cannot fails and hence we can generate p. Thanks to Lemma A.4, we know that we can divide $W_\mathcal{T}$ into $\widetilde{W_\mathcal{T}}$, that are the fields that have to be id in all the transformations of Y, and $W_\mathcal{T} \setminus \widetilde{W_\mathcal{T}}$, the fields that can be associated with any constant transformation λ_a. Unless there exists a field w for which no possible value a_w is such that the predicates verified by a are X_{2_w}, we can thus choose an appropriate value for each t_w such that $w \in W_\mathcal{T} \setminus \widetilde{W_\mathcal{T}}$. If it is not possible, then `TAKE_ONE` fails and ω_{X_1,Y,X_2} is empty. For each field $w \in \widetilde{W_\mathcal{T}}$, t_w has to be id. It is true that $g(t(p)) = X_2$ if and only if $\forall w \in W. g(t(p))_w = X_{2_w}$. For $w \in \widetilde{W_\mathcal{T}}$ this is the same as $X_{1_w} = X_{2_w}$, which is checked by Algorithm 5, that fails if it does not holds. □

A.5 Firewall Expressivity: Proofs

Lemma A.5 (IFCL universality)**.** For each function $\tau : \mathbb{P} \to \mathcal{T}_\mathbb{P}$ there exists an IFCL firewall $\mathcal{F}$ such that $(\!|\mathcal{F}|\!) = \tau$.

Proof. Trivial: just take a control diagram with a single node q and a configuration such that $f(q) = \tau$. □

As stated in subsection 2.6.1, we only consider firewalls where no packets cycle. Then we define the following function, that simplifies $\odot$ by removing cycle detection.

$$\otimes^{(C,f)}(q)(p) = \begin{cases} \otimes^{(C,f)}(q')(p') \circ t & \text{if } q \neq q_f \wedge t \neq \lambda_\perp \\ t & \text{if } q = q_f \vee t = \lambda_\perp \end{cases}$$

with $t = f(q)(p)$, $p' = t(p)$ and $q' = \delta(q, p')$.

Now we state the following general property.

Lemma A.6. Let (C, f) be a firewall, then

$$\forall p \in \mathbb{P}.\ \odot_{\{q_i\}}^{(C,f)}(q)(p) = \lambda_{\circlearrowleft} \vee \odot_{\{q_i\}}^{(C,f)}(q)(p) = \otimes^{(C,f)}(q)(p)$$

Proof. We prove the more general statement

$$\forall I \subseteq Q. \forall p \in \mathbb{P}.\ \odot_{I}^{(C,f)}(q)(p) = \lambda_{\circlearrowleft} \vee \odot_{I}^{(C,f)}(q)(p) = \otimes^{(C,f)}(q)(p)$$

The proof proceed by induction on $\odot$. If $q = q_f$ or $t = \lambda_\perp$, then the thesis trivially holds. Assume $q \neq q_f$ and $t \neq \lambda_\perp$, let $t' = f(q)(p)$, $p' = t'(p)$ and $q' = \delta(q, p')$, if $q' \in I$ then the premise is false and the thesis holds. The last case is when $q' \notin I$ and $q \neq q_f$, then by definition $\otimes^{(C,f)}(q)(p) = \otimes^{(C,f)}(q')(p') \circ t'$ and $\odot_{I}^{(C,f)}(q)(p) = \odot_{I \cup \{q'\}}^{(C,f)}(q')(p') \circ t'$. By induction hypothesis, $\forall J \subseteq Q$, $\forall p \in \mathbb{P}$, $\odot_{J}^{(C,f)}(q')(p) = \lambda_{\circlearrowleft} \vee \odot_{J}^{(C,f)}(q')(p) = \otimes^{(C,f)}(q')(p)$. Take $J = I \cup \{q'\}$, if $\odot_{I \cup \{q'\}}^{(C,f)}(q')(p') = \lambda_{\circlearrowleft}$, then $\odot_{I}^{(C,f)}(q)(p) = \lambda_{\circlearrowleft}$ and the thesis follows. Otherwise, $\otimes^{(C,f)}(q')(p') = \odot_{I}^{(C,f)}(q')(p')$ and the thesis follows. □

To prove Theorem 2.5 we use the following lemma.

Lemma A.7. Let $C = (Q, A, q_i, q_f)$ be a control diagram, let V be a cap-label assignment, then the two following condition are equivalent

$$\begin{aligned}
&(i) \quad \exists f \text{legal for V. } \otimes^{(C,f)}(q)(p) = t \\
&(ii) \quad \exists(\pi, v).\ \pi = q_1 \cdot \ldots q_n \wedge v = l_1 \cdot \ldots l_n \\
&\qquad \wedge\ \forall j.\ \exists \psi.\ (q_j, \psi, q_{j+1}) \in A \wedge l_j \in V(q_j) \\
&\qquad \wedge\ \forall j \neq n.\ q_j \neq q_f \wedge l_j \neq \texttt{DROP} \\
&\qquad \wedge\ \forall i, j.\ i \neq j \Rightarrow q_i \neq q_j \\
&\qquad \wedge\ q_1 = q \wedge (q_n = q_f \vee l_n = \texttt{DROP}) \wedge \\
&\qquad \exists t_1, t_2, \ldots t_n.\ \forall j.\ t_j \in \varepsilon(l_j) \wedge \\
&\qquad\quad t_n \circ t_{n-1} \cdots \circ t_1 = t \wedge \\
&\qquad\quad \forall j < n.\ \psi_j(p_j)
\end{aligned}$$

Proof. We prove separately $(i) \Rightarrow (ii)$ and $(ii) \Rightarrow (i)$.
For $(i) \Rightarrow (ii)$ we proceed for induction on the calls of function $\otimes$. We assume the premise and consider the following exhaustive cases. If $q = q_f$ then $t \in \epsilon l$ for

some $l \in V(q_f)$ and we take $\pi = q_f$ and $v = l$: (ii) trivially holds. Otherwise if $f(q)(p) = \lambda_\perp$, since f is legal for V then $\texttt{DROP} \in V(q)$; we take $\pi = q$ and $v = \texttt{DROP}$ for which (ii) trivially holds. Finally, assume $q \neq q_f$, $f(q)(p) = t' \neq \lambda_\perp$ and $\delta(q, p') = q'$ where $p' = t'(p)$, then $\otimes^{(C,f)}(q)(p) = t$ is equal to $\otimes^{(C,f)}(q')(p') \circ t'$. From the induction hypothesis we have that exists a pair (π', v') such that all the conjuncts in (ii) hold for it. Since f is legal for V we know that a cap-label $l \in V(q)$ must be such that $f(q)(p) \in \varepsilon(l)$. We build $\pi = q \cdot \pi'$ and $v = l \cdot v'$. By the induction hypothesis and by construction $\forall j. \exists \psi. (q_j, \psi, q_{j+1}) \in A$, since $\delta(q, p') = q'$; also it holds that $l_j \in V_{\mathcal{L}}(q_j)$. The condition $\forall j \neq n. q_j \neq q_f \wedge l_j \neq \texttt{DROP}$ holds by hypothesis on q and by the induction hypothesis for the rest of π. For construction $q_1 = q$ and by the induction hypothesis $(q_n = q_f \vee l_n = \texttt{DROP})$. $\psi(p')$ holds because of $\delta(q, p') = q'$. Finally $t_n \circ t_{n-1} \cdots \circ t_1 = t$ holds because the the induction hypothesis guarantee $t_n \circ t_{n-1} \cdots \circ t_2 = t''$ such that $t'' \circ t' = t$.

We then show that $(ii) \Rightarrow (i)$ holds. We take f such that $\forall j. f'(q_j)(p) = t_j$ and $\forall q \notin \pi. f(q)(p) = t$ for whichever $t \in \varepsilon(l)$ with l assigned to q. Hence, by construction f is legal for V; $\otimes^{(C,f)}(q)(p) = t$ is now proved by induction on the length n of π and v. If $n = 1$ then either $q_1 = q_f$ or $l_1 = \texttt{DROP}$. In the first case then $t \in \epsilon l$ for some $l \in V(q_f)$ and also $\otimes^{(C,f)}(q)(p) = t$. Otherwise if $l_1 = \texttt{DROP}$ then $f(q)(p) = t_1 = \lambda_\perp$. Finally suppose that the statement holds for any $\pi' = q' \cdot \pi''$ and v' of length n, and take $(\pi, v) = (q \cdot \pi', l \cdot v')$ of length $n + 1$. By the induction hypothesis and (ii) we have that $\otimes^{(C,f)}(q')(p') = t''$ where $t' = f(q)(p)$, $p' = t'(p)$ and $t = t'' \circ t'$. □

Theorem 2.5. $E_{\mathcal{L}} = \widetilde{E}_{\mathcal{L}}$.

Proof. Follows trivially from Lemma A.6 and A.7. □

Lemma A.8. Given a trace $h = (\pi, v)$ and a packet p, the following are equivalent

$$
\begin{array}{ll}
(i) & \text{CHECK\_FLOW}(h, p) \\
(ii) & \exists p_1, \ldots, p_n. p_1 = p \wedge \\
 & \qquad \forall j < n - 1. \psi_j(p_j) \wedge \\
 & \qquad \forall j < n. \exists t_j \in \varepsilon(l_j). t_j(p_j) = p_{j+1}
\end{array}
$$

Proof. Formula (ii) is equivalent to the following in which packet fields are made explicit:

$$
\begin{array}{l}
\exists p_{1_{w_1}}, \ldots, p_{n_{w_1}}, p_{1_{w_2}}, \ldots, p_{n_{w_2}}, \ldots p_{1_{w_m}}, \ldots, p_{n_{w_m}}. \\
\quad p_{1_{w_1}} = p_{w_1} \wedge \ldots p_{1_{w_m}} = p_{w_m} \wedge \\
\quad \forall j < n - 1. \psi_{j_{w_1}}(p_{j_{w_1}}) \wedge \cdots \wedge \psi_{j_{w_m}}(p_{j_{w_m}}) \wedge \\
\quad \forall j < n. \exists t_j \in \varepsilon(l_j). t_j(p_j)_{w_1} = p_{j+1_{w_1}} \wedge \cdots \wedge t_j(p_j)_{w_m} = p_{j+1_{w_m}}
\end{array}
$$

We can then substitute

$$
\exists t_j \in \varepsilon(l_j). t_j(p_j)_{w_1} = p_{j+1_{w_1}} \wedge \cdots \wedge t_j(p_j)_{w_m} = p_{j+1_{w_m}}
$$

with $\forall w \notin \gamma(l_j).\ p_{j_w} = p_{j+1_w}$

We omit the constraints on $w \in \gamma(l_j)$ because any value can be arbitrarily chosen by t_j for the fields in $\gamma(l_j)$. Substitution for constraints on $w \in \gamma(l_j)$ is legal because, for every label l, every transformation $t \in \varepsilon(l)$ and every field $w \in \gamma(l_j)$, $t_w = id$.

We can then replace p for every occurrence of p_1 and remove the existential quantification on its fields, since $p = p_1$. Finally, for each pair of packet fields such that $p_{i_w} = p_{j_w}$ with $i < j$ we instantiate p_{j_w} to p_{i_w}, removing the existential quantification on p_{j_w}. We repeat the last step until we reach a fixpoint where no further reduction is possible. The formula obtained is the conjunction of a predicate on the packet p ($(ii)_A$) and an existentially quantified predicate on the fields of some intermediate packets ($(ii)_B$).

The lemma holds because, for any iteration, $(ii)_A$ is true iff $Ext(\psi_j, CL)(p)$ at line 8 of CHECK_FLOW(h, p) is true, and $(ii)_B$ is true iff $Sat(\overline{\psi})$ at line 8 of CHECK_FLOW(h, p) is true. The first coimplication trivially holds because, for every j, the use of *Ext* with CL excludes all and only the conjuncts ψ_{j_w} of ψ_j that predicate on existentially quantified field values that do not relate to p. To show that the second coimplication holds, consider $(ii)_B$. Note that for each field w, the instantiation above partitions the constraints on existentially quantified field-values in disjoint intervals of indexes. Those constraints are the same accumulated by $\overline{\psi}$, hence the thesis holds since asking for $\overline{\psi}$ to be satisfiable is exactly the same as asking for constraint-satisfying field-values to exists. □

Theorem 2.6. Given a trace $h = (\pi, v)$, the pair (p, t) is in $\tilde{E}_h$ iff

$$t \in \hat{\varepsilon}(v) \wedge \text{CHECK\_FLOW}(h, p) \wedge \text{CHECK\_FLOW}(\text{REV}(h), t(p))$$

Proof. $(p, t) \in \tilde{E}_h$ implies by definition $t \in \hat{\varepsilon}(v)$. We then establish the following, assuming $t \in \hat{\varepsilon}(v)$

$$\begin{aligned}
(a)\ & \exists p_1, \ldots, p_{n+1}.\ p_1 = p \wedge \\
& \quad \forall j < n.\ \psi_j(p_j) \wedge \\
& \quad \forall j \leq n.\ \exists t_j \in \varepsilon(l_j).\ t_j(p_j) = p_{j+1} \wedge \\
(b)\ & \exists p'_1, \ldots, p'_{n+1}.\ p'_{n+1} = t(p) \wedge \\
& \quad \forall j < n.\ \psi_j(p'_j) \wedge \\
& \quad \forall j \leq n.\ \exists t'_j \in \varepsilon(l_j).\ t'_j(p'_j) = p'_{j+1}
\end{aligned}$$

is equivalent to

$$\begin{aligned}
(c)\ & \exists p''_1, \ldots, p''_{n+1}.\ p''_1 = p \wedge p''_{n+1} = t(p) \wedge \\
& \quad \forall j < n.\ \psi_j(p''_j) \wedge \\
& \quad \forall j \leq n.\ \exists t''_j \in \varepsilon(l_j).\ t''_j(p''_j) = p''_{j+1}
\end{aligned}$$

$(c) \Rightarrow (a) \wedge (b)$ holds trivially. Now assume (a) and (b) and we build $p''_1, \ldots, p''_{n+1}$ such that $p''_1 = p \wedge p''_{n+1} = t(p)$ satisfying (c). It is sufficient to take, for each

j, $t''_{j_w} = t'_{j_w}$ if $t_w \neq id$ and id otherwise. Then, the thesis follows trivially by Lemma A.8. □

Before proving Theorem 2.7 we prove two auxiliary lemmata.

Lemma A.9. For any trace h and any pair of packets p, p' that satisfy the same set of predicates, i.e. such that $g(p) = g(p')$, the following holds

$$\text{CHECK\_FLOW}(h, p) \Leftrightarrow \text{CHECK\_FLOW}(h, p')$$

Proof. The statement follows trivially by definition since the p parameter of CHECK_FLOW in Algorithm 3 is only used when checking if it verifies the predicates of the trace. □

Lemma A.10. For each trace (π, v), $\hat{\varepsilon}(v)$ is in $\mathbb{T}$.

Proof. We will prove separately that

$$\begin{aligned}
&(i) \quad \hat{\varepsilon}(l \cdot \texttt{ID}) = \hat{\varepsilon}(\texttt{ID} \cdot l) = \varepsilon(l) \\
&(ii) \quad \hat{\varepsilon}(\texttt{SNAT} \cdot \texttt{DNAT}) = \hat{\varepsilon}(\texttt{DNAT} \cdot \texttt{SNAT}) = \Lambda \times \Lambda \\
&(iii) \quad \hat{\varepsilon}(v) = \Lambda \times \Lambda \wedge l \neq \texttt{DROP} \Longrightarrow \hat{\varepsilon}(v \cdot l) = \hat{\varepsilon}(l \cdot v) = \Lambda \times \Lambda \\
&(iv) \quad \hat{\varepsilon}(v \cdot \texttt{DROP}) = \hat{\varepsilon}(\texttt{DROP} \cdot v) = \varepsilon(\texttt{DROP})
\end{aligned}$$

Item (i) holds trivially since $\varepsilon(\texttt{ID}) = \{id\}$. For (ii) note that $\hat{\varepsilon}(\texttt{SNAT} \cdot \texttt{DNAT}) = \varepsilon(\texttt{SNAT}) \circ \varepsilon(\texttt{DNAT})$; take $t \in \hat{\varepsilon}(\texttt{SNAT} \cdot \texttt{DNAT})$, $t = t' \circ t''$ with $t' \in \varepsilon(\texttt{SNAT})$ and with $t'' \in \varepsilon(\texttt{DNAT})$. By contradiction assume $t_{sIP} = t_{sPort} = id$, then $t'_{sIP} = t'_{sPort} = id$, hence $t' \notin \varepsilon(\texttt{SNAT})$; same for DNAT. (iii) holds because id is the identity of $\circ$ and because $\lambda_a \circ \lambda_{a'} = \lambda_a$. (iv) holds by definition of $\hat{\epsilon}$ and $\lambda_\perp$. □

Theorem 2.7. Ω is a partition of $\mathbb{P} \times \mathcal{T}_\mathbb{P}$ such that the elements of ω_{X_1,Y,X_2} are either all expressible or all not expressible.

Proof. We first prove that Ω is a partition, by separately establishing the two following statements

$$\begin{aligned}
&(i) \quad \forall (p,t) \in \mathbb{P} \times \mathcal{T}_\mathbb{P}.\, \exists X_1, Y, X_2.\, (p,t) \in \omega_{X_1,Y,X_2} \\
&(ii) \quad (X_1, Y, X_2) \neq (X'_1, Y', X'_2) \wedge \omega_{X_1,Y,X_2} \neq \omega_{X'_1,Y',X'_2} \\
&\qquad\qquad \Rightarrow \omega_{X_1,Y,X_2} \cap \omega_{X'_1,Y',X'_2} = \emptyset
\end{aligned}$$

For (i), take a pair (p,t). If $t = \lambda_\perp$ then the thesis follows by construction with $X_1 = g(p)$, $Y = \varepsilon(\texttt{DROP})$ and any X_2. Otherwise take $X_1 = g(p)$ and $X_2 = g(t(p))$; also if $t = id$ take $Y = \varepsilon(\texttt{ID})$; if $t_{dIP} = t_{dPort} = id$ take $Y = \varepsilon(\texttt{SNAT})$; if $t_{sIP} = t_{sPort} = id$ take $Y = \varepsilon(\texttt{DNAT})$; in the other cases take $Y = \Lambda \times \Lambda$.

For (ii) assume that $X_1 \neq X'_1$, and by contradiction that $(p,t) \in \omega_{X_1,Y,X_2} \cap \omega_{X'_1,Y',X'_2}$. Then by definition of ω_{X_1,Y,X_2} one has $X_1 = g(p) = X'_1$. Instead, assume $Y \neq Y'$ and that $(p,t) \in \omega_{X_1,Y,X_2} \cap \omega_{X'_1,Y',X'_2}$, then by definition of ω_{X_1,Y,X_2} we know that $t \in Y$ and $t \in Y'$, but $\mathbb{T}$ is trivially a partition of $\mathcal{T}_\mathbb{P}$, hence $Y = Y'$. Finally, assume $X_2 \neq X'_2$: if $Y = \varepsilon(\texttt{DROP})$ we get that

$\omega_{X_1,Y,X_2} = \omega_{X_1',Y',X_2'}$, contradicting the hypothesis; otherwise let (p,t) be $\in \omega_{X_1,Y,X_2} \cap \omega_{X_1',Y',X_2'}$, then $X_2 = g(t(p)) = X_2'$ holds.

Now we prove that the elements of ω_{X_1,Y,X_2} are either all expressible or all not expressible. By Theorem 2.5 and 2.6, it suffices proving the following, for each trace $h = (\pi, v)$, and for each $(p,t), (p',t') \in \omega_{X_1,Y,X_2}$

$$t \in \hat{\varepsilon}(v) \wedge \text{CHECK\_FLOW}(h,p) \wedge \text{CHECK\_FLOW}(\text{REV}(h), t(p))$$
$$\Leftrightarrow$$
$$t' \in \hat{\varepsilon}(v) \wedge \text{CHECK\_FLOW}(h,p') \wedge \text{CHECK\_FLOW}(\text{REV}(h), t'(p'))$$

We prove the following stronger statements

(a) $t \in \hat{\varepsilon}(v) \Leftrightarrow t' \in \hat{\varepsilon}(v)$

(b) $\text{CHECK\_FLOW}(h,p) \Leftrightarrow \text{CHECK\_FLOW}(h,p')$

(c) $\text{CHECK\_FLOW}(\text{REV}(h), t(p)) \Leftrightarrow \text{CHECK\_FLOW}(\text{REV}(h), t'(p'))$

From Lemma A.10 we know that $\hat{\varepsilon}(v)$ is one of the equivalence classes in $\mathbb{T}$, hence (a) holds because $t, t' \in \hat{\varepsilon}(v) \Leftrightarrow Y = \hat{\varepsilon}(v)$, where Y indexes the considered ω class. To prove (b) ((c), resp.) it suffices to apply Lemma A.9 to h (REV(h), resp.). □

Theorem 2.8. $E_{\texttt{pf}} = E_{\texttt{ipfw}} \subsetneq E_{\texttt{iptables}} = E_{IFCL} = \mathbb{P} \times \mathcal{T}_{\mathbb{P}}$

Proof. The thesis trivially follows from Table 2.3. □

Theorem 2.9. Given a language $\mathcal{L}$ and a fw-function τ

$$\tau \in T_{\mathcal{L}} \text{ only if } \forall p \in \mathbb{P}.\, (p, \tau(p)) \in E_{\mathcal{L}}.$$

Proof. By contradiction assume that $\exists p.$ (i) $\tau \in T_{\mathcal{L}}$, and (ii) $(p, \tau(p)) \notin E_{\mathcal{L}}$. Item (i) implies by definition that $\exists f$ legal for $V_{\mathcal{L}}$ is such that $(\!|(C_{\mathcal{L}}, f)|\!) = \tau$, and hence $\forall p \in \mathbb{P}.\, (\!|(C_{\mathcal{L}}, f)|\!)(p) = \tau(p)$. Finally, item (ii) is equivalent to $\nexists f$ legal for $V_{\mathcal{L}}$ such that $(\!|(C_{\mathcal{L}}, f)|\!)(p) = \tau(p)$. Contradiction. □

Corollary A.1. $E_{\mathcal{L}} \nsubseteq E_{\mathcal{L}'} \Rightarrow T_{\mathcal{L}} \nsubseteq T_{\mathcal{L}'}$

Theorem 2.10.

- $T_{\texttt{pf}} \subsetneq T_{\texttt{ipfw}} \subsetneq T_{IFCL}$
- $T_{\texttt{iptables}} \subsetneq T_{IFCL}$
- $T_{\texttt{pf}} \nsubseteq T_{\texttt{iptables}}$, $T_{\texttt{iptables}} \nsubseteq T_{\texttt{pf}}$
- $T_{\texttt{ipfw}} \nsubseteq T_{\texttt{iptables}}$, $T_{\texttt{iptables}} \nsubseteq T_{\texttt{ipfw}}$

Proof. IFCL dominates all the other languages by Lemma A.5. Moreover, by Theorem 2.8 and Corollary A.1, the following holds trivially:

$$T_{\texttt{iptables}} \nsubseteq T_{\texttt{ipfw}} \qquad T_{\texttt{iptables}} \nsubseteq T_{\texttt{pf}}$$

Then we show that (i) $T_{\texttt{pf}} \subseteq T\_\texttt{ipfw}$, we exhibit (ii) a function τ_1 that is in $T_{\texttt{ipfw}}$ but not in $T_{\texttt{pf}}$ and (iii) a function τ_2 that is in $T_{\texttt{pf}}$ but not in $T_{\texttt{iptables}}$.

For proving (i) let f be a configuration legal for $V_{\texttt{pf}}$. Then we build an equivalent configuration f' legal for $V_{\texttt{ipfw}}$ such that $(\!|(C_{\texttt{pf}}, f)|\!) = (\!|(C_{\texttt{ipfw}}, f')|\!)$, by taking $f'(q_0) = f(q_1) \circ f(q_0)$ and $f'(q_1) = f(q_3) \circ f(q_2)$ (note that in the control diagram of `ipfw`, the cap-labels of q_1 are the union of the cap-labels of q_2 and q_3 in the control diagram of `pf`).

To establish (ii), take the following function τ_1, where $b, b' \in \mathcal{S}$ and $c \notin \mathcal{S}$.

$$\tau_1(p) = \begin{cases} (id : id, \lambda_{b'} : id) & \text{if } p_{dIP} = c \wedge p_{sIP} = b \\ \lambda_\perp & \text{if } (p_{dIP} = c \wedge p_{sIP} = b') \vee \\ & \quad p_{dIP} \in \mathcal{S} \vee p_{sIP} \notin \mathcal{S} \\ id & \text{otherwise} \end{cases}$$

The function τ_1 is in $T_{\texttt{ipfw}}$, indeed $(\!|(C_{\texttt{pf}}, f)|\!) = \tau_1$ for f as follows.

$$f(q_0)(p) = \lambda_\perp$$

$$f(q_1)(p) = \begin{cases} (id : id, \lambda_{b'} : id) & \text{if } p_{dIP} = c \wedge p_{sIP} = b \\ \lambda_\perp & \text{if } (p_{dIP} = c \wedge p_{sIP} = b') \vee \\ & \quad p_{dIP} \in \mathcal{S} \\ id & \text{otherwise} \end{cases}$$

To show that $\tau_1 \notin T_{\texttt{pf}}$, by contradiction suppose that there is a configuration f' legal for $V_{\texttt{pf}}$ and such that $(\!|(C_{\texttt{pf}}, f')|\!) = \tau_1$. Consider two packets p and p' with source b and b' resp., and the same destination c, which both traverse the nodes q_0 and q_1 of `pf`. It must be $f'(q_0)(p) = (id : id, \lambda_{b'} : id)$, and $f'(q_0)(p') = id$, transforming the two packets in the same packet p'' with source b' and destination c. Also it must be $f'(q_1)(p'') = id$ and at the same time that $f'(q_1)(p'') = \lambda_\perp$, because τ_1 keeps p and p' apart: contradiction.

For proving (iii), take the following function τ_2, where a, a', a'', b and b' are in $\mathcal{S}$.

$$\tau_2(p) = \begin{cases} (\lambda_{a'} : id, \lambda_{b'} : id) & \text{if } p_{dIP} = a \wedge p_{sIP} = b \\ (\lambda_{a'} : id, id : id) & \text{if } p_{dIP} = a \wedge p_{sIP} = b' \\ (\lambda_{a'} : id, id : id) & \text{if } p_{dIP} = a'' \wedge p_{sIP} = b \\ \lambda_\perp & \text{otherwise} \end{cases}$$

The function τ_2 is in $T_{\texttt{pf}}$, indeed $(\!|(C_{\texttt{pf}}, f)|\!) = \tau_2$ for f as follow.

$$f(q_0)(p) = \begin{cases} (id : id, \lambda_{b'} : id) & \text{if } p_{dIP} = a \wedge p_{sIP} = b \\ id & \text{otherwise} \end{cases}$$

$$f(q_1)(p) = \begin{cases} id & \text{if } p_{dIP} = a \wedge p_{sIP} = b' \\ id & \text{if } p_{dIP} = a'' \wedge p_{sIP} = b \\ \lambda_\perp & \text{otherwise} \end{cases}$$

$$f(q_2)(p) = \begin{cases} (\lambda_{a'} : id, id : id) & \text{if } (p_{dIP} = a \wedge p_{sIP} = b') \\ & \quad \vee (p_{dIP} = a'' \wedge p_{sIP} = b) \\ id & \text{otherwise} \end{cases}$$

$$f(q_3)(p) = \begin{cases} id & \text{if } p_{dIP} = a' \wedge \\ & \quad (p_{sIP} = b \vee p_{sIP} = b') \\ \lambda_\perp & \text{otherwise} \end{cases}$$

To show that $\tau_2 \notin T_{\texttt{iptables}}$, by contradiction suppose that there is a configuration f' legal for $V_{\texttt{iptables}}$ and such that $(\!|\, (C_{\texttt{iptables}}, f') \,|\!) = \tau_2$. Consider two packets p and p' with the same source b and with destination a and a'', resp. They both traverse the nodes q_8 and q_5. It must be $f'(q_8)(p) = f'(q_8)(p') = (\lambda_{a'} : id, id : id)$, transforming the two packets in the same packet p'' with source b and destination a'. Also it must be $f'(q_5)(p'') = id$ and at the same time that $f'(q_5)(p'') = (id : id, \lambda_{b'} : id)$, because τ_2 keeps p and p' apart: contradiction. □

We formalize the assumptions of subsection 2.6.5.

Assumption A.1. For every firewall language $\mathcal{L}$

1. Algorithm 4 does not consider any trace (π, v) with $\pi = (q_1, \dots q_n)$ and $v = (l_1, \dots, l_n)$ where $l_n = \texttt{DROP}$ and $\exists i.\, i < n$ such that either $\texttt{DROP} \in V(q_i)$ or $l_i \neq \texttt{ID}$;

2. we do not consider configurations f such that, for some p, $(\!|\, (C_\mathcal{L}, f) \,|\!)(p) = \lambda_\perp$ and $\neg\chi_{q_i}(p)$, where

$$\chi_q(p) = \begin{cases} true & \text{if } q = q_f \\ (\texttt{DROP} \in V_\mathcal{L}(q) \wedge p \neq \perp \Rightarrow t = \lambda_\perp) \wedge & \\ \qquad t \in \{id, \lambda_\perp\} \wedge \chi_{\delta(q,t(p))} t(p) & \text{o.w.} \end{cases}$$

   with $t = f(q)(p)$;

3. every pair (p, t) is expressed by at most one trace h in $\mathcal{H}_\mathcal{L}$.

4. The control diagram $C_\mathcal{L}$ is such that no trace $h \in \mathcal{H}_\mathcal{L}$ contains repetitions of $\texttt{SNAT}$ or $\texttt{DNAT}$ cap-labels.

Lemma A.11. For any language $\mathcal{L}$, legal configuration f, packet p and transformation t, (i) $(\!|\, (C_\mathcal{L}, f) \,|\!)(p) = t$ iff

$$\begin{aligned} (ii) \quad & \exists (q_1, \dots, q_n, l_1, \dots, l_n) \in \mathcal{H}_\mathcal{L}. \\ & \exists\, t_1, \dots t_n.\ t_n \circ \dots \circ t_1 = t \wedge \\ & \forall j.\ f(q_j)(p_j) = t_j \ \wedge\ j < n \Rightarrow \psi_j(p_{j+1}) \end{aligned}$$

where $p_1 = p$ and $\forall j.\ p_{j+1} = (t_j \circ \dots \circ t_1)(p)$.

Algorithm 6 Single packet version of Algorithm 4.

```
1: function CHECK_FUNCTION(τ, C, V)
2:     for all q ∈ Q do g(q) ← ∅
3:     for all (p, t) ∈ τ do
4:         h ← COMPUTE_TRACE(C, V, (p, t))
5:         if h = Null then print (p, t) not expressible
6:         else g ← CHECK_PAIR(h, (p, t), g)

7: function COMPUTE_TRACE(C, V, (p, t))
8:     for all h ∈ H_L do
9:         if t ∈ ε̂(v) ∧ CHECK_FLOW(h, p)∧
             CHECK_FLOW(REV(h), t(p)) then return h
10:    return Null

11: function CHECK_PAIR(h, (p, t), g)
12:    (p@, t▷, t◁) ← (p, t, (id : id, id : id))
13:    for all (q, l) ∈ h do
14:        (t@, t▷) ← SPLIT(t▷, l)
15:        for all ((p̃@, t̃◁, t̃@), (p̃, t̃)) ∈ g(q) s.t. t̃@ ≠ t@ do
16:            if p@ = p̃@ then
17:                print (p, t) and (p̃, t̃) clash in q : (p@, t@, t̃@)
18:        g(q) ← g(q) ∪ {((p@, t◁, t@), (p, t))}
19:        t◁ ← t@ ∘ t◁
20:        p@ ← t@(p@)
21:    return g
```

Proof. Trivially follows from the definition of $\otimes$ and Lemma A.6. As for Lemma A.7, we can proceed for induction, on the calls of function $\otimes$ for $(i) \Rightarrow (ii)$, and on n for $(ii) \Rightarrow (i)$. □

Theorem 2.11. For each firewall language $\mathcal{L}$ and fw-function τ, the Algorithm 4 is correct because it prints all and only

1. the τ-pairs (P, t) not expressible by $\mathcal{L}$;
2. the τ-pairs (P, t) and $(\tilde{P}, \tilde{t})$ that clash on some node q.

Proof. We start with item 1), and we note that Algorithm 4 takes $([P], t)$ out of each of the τ-pairs $(P, t) \subseteq \omega \in \Omega$, which by Theorem 2.7 faithfully represents them all. By Theorem 2.6, checking expressivity for a trace is the same of using CHECK_FLOW as done by function COMPUTE_TRACE. Finally, Theorem 2.5 reduces the expressivity of a language to that of its traces.

For proving item 2), we simplify Algorithm 4 to operate on a single pair (p, t), obtaining Algorithm 6; then the statement follows trivially. Consider

Lemma A.11 and an expressible pair (p, t). Because of Property 3, we know there is only one trace h such that

$$\exists\, t_1, \dots t_n.\\ t_n \circ \dots \circ t_1 = t \wedge \forall j.\ f(q_j)(p_j) = t_j \wedge (j < n \Rightarrow \psi_j(p_{j+1}))$$

It is trivial to prove that, because of Property 4, for each expressible pair (p, t) and trace h, there is only one legal way of decomposing t in such $t_1, \dots t_n$. Hence $(\!|\, (C_{\mathcal{L}}, f) \,|\!)(p) = t$ iff

$$\forall j.\ f(q_j)(p_j) = t_j \ \wedge\ (j < n \Rightarrow \psi_j(p_{j+1}))$$

with h returned by COMPUTE_TRACE, and $t_1, \dots t_n$ unique legal decomposition of t.

Note that, as h and $t_1, \dots t_n$ are uniquely determined, we can show that $\forall j.\ f(q_j)(p_j) = t_j \Rightarrow (j < n \Rightarrow \psi_j(p_{j+1}))$. Indeed, the opposite would imply that (p, t) is not expressible, leading to a contradiction. Hence, we can finally conclude that $(\!|\, (C_{\mathcal{L}}, f) \,|\!)(p) = t$ iff $\forall j.\ f(q_j)(p_j) = t_j$ with h returned by COMPUTE_TRACE, and $t_1, \dots t_n$ unique legal decomposition of t. To show the correctness of Algorithm 6 it is then sufficient to prove that the list of transformations $t_@$ generated by Algorithm 6 is a legal decomposition of t, which is true by definition of SPLIT. □

Corollary 2.1. A fw-function τ is expressible by $\mathcal{L}$ if and only if Algorithm 4 prints nothing.

Proof. We show that τ is expressible by $\mathcal{L}$ (i.e. $\tau \in T_{\mathcal{L}}$) iff

1. all pairs $(p, \tau(p))$ are expressible by $\mathcal{L}$;
2. no pairs $(p, \tau(p))$ and $(\tilde{p}, \tau(\tilde{p}))$ collide on some node q.

$\tau \in T_{\mathcal{L}} \Rightarrow (1), (2)$ trivially holds. Conversely, assume (1) and (2). For each pair $(p, \tau(p))$ there is only one possible trace and decomposition of t. Thus, we can accumulate the set of conditions on f, that are necessary and sufficient to have $(\!|\, (C_{\mathcal{L}}, f) \,|\!)(p) = \tau(p)$. Each of these conditions states that $f(q_@)(p_@) = t_@$ for some $q_@$, $p_@$ and $t_@$. Hence, for any $p \in \mathbb{P}$ we build a set F_p of triplets $(q_@, p_@, t_@)$ such that, for any legal configuration f

$$(\!|\, (C_{\mathcal{L}}, f) \,|\!)(p) = \tau(p) \Longleftrightarrow \forall (q_@, p_@, t_@) \in F_p.\ f(q_@)(p_@) = t_@$$

We build then the following configuration f:

$$\forall q, p.\ f(q)(p) = \begin{cases} t & \text{if } (q, p, t) \in \bigcup_{p \in \mathbb{P}} F_p \\ id & \text{o.w.} \end{cases}$$

Note that f is indeed a function because we assume there are no clash. □

Appendix B

Technical Details and Proofs of Chapter 3

B.1 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 $p, p', \ldots$ 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.

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

$$\text{(N-3)}\ \frac{\overline{eval}_{\sigma;\#}(m) = m'}{(\sigma, call\ m([a])) \to (\sigma, call\ m'([a]))} \qquad \text{(N-4)}\ \frac{(\sigma, call\ m([a])) \in \Gamma \quad (m, d) \in \Gamma}{add(\sigma, d)}$$

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

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

Figure B.1: CIL normalization rules.

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

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

$$\text{(S-ta2)}\ \frac{(B_\#, typeattributeset\ a\ (expr))) \in \Gamma \quad (B'_\#, type\ n) \in \Gamma \quad [\![expr]\!]_\Gamma(B'_\#.t)}{(a, B'_\#.t) \in ta}$$

$$\text{(S-A1)}\ \frac{(B_\#, allow\ t\ t'\ (class\ (perms))) \in \Gamma}{(t, perms, t') \in A}$$

$$\text{(S-A2)}\ \frac{(t1, perms, t2) \in A \quad t1' \in ta(t1) \quad t2' \in ta(t2)}{(t1', perms, t2') \in A}$$

Figure B.2: CIL semantics.

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 $\perp$ is resolution is not possible in σ.

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

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 $\perp$, 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

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

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 $\perp$, and evaluates it in σ' otherwise.

The CIL normalization pipeline consists of the six rewriting rules in Figure B.1, 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 in 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 B.2, which yield the graph $G = (N, ta, A)$. Since attribute expressions $expr$ are indeed boolean functions on types and typeattributes, we assume a denotational semantics $[\![expr]\!] : N \to \{\text{true}, \text{false}\}$.

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

$$\text{(comp)}\ \frac{P_1 \preceq_P P_1' \quad P_2 \preceq_P P_2'}{P_1 P_2 \preceq_P P_1' P_2'} \qquad \text{(P-1)}\ \frac{n_1 \preceq_n n_1' \quad n_2 \preceq_n n_2' \quad oa \preceq_{oa} oa'}{n_1 \ oa \ n_2 \preceq_P n_1' \ oa' \ n_2'}$$

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

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

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

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

Figure B.3: Definition of IFL requirement refinement and auxiliary relations.

B.2 IFL: Proofs

IFL requirement refinement is defined by the $\preceq$ operator in Figure B.3, where reflexivity and transitivity rules are implicitly assumed for every defined relation, and where arrows $>$ or $+>$ 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 B.1 (Kind refinement)**.** Let I be an information flow diagram, if $P \preceq P'$ then for all path π in I, $\pi \triangleright_I P$ implies $\pi \triangleright_I P'$

Proof. Assume $\pi \triangleright_I P$ and not $\pi \triangleright_I P'$ and proceeds by proving for each rule in Figure B.3 and for arbitrary π that if the lemma holds on the premises, that it also does on the conclusion.

Consider rule (comp), and let $i \in \{1, 2\}$. From $\pi_i \triangleright_I P_i$ we have that $\pi_i \triangleright_I P_i'$. Then $\pi_1 \pi_2 \triangleright_I P_1' P_2'$ by the definition of $\triangleright$.

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

previous case, and the second one requires induction on the occurrences of the previous case (note that $* \preceq *$).

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

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

The proof of rule (P-4) is almost the same of the one of (P-2).

□

Theorem 3.1 (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

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

Proof. We consider the rules in Figure B.3, and for arbitrary I we show that if the premises are met, then the theorem holds for the conclusion.

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

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

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

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

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

$$\text{(N-2')}\ \frac{\begin{array}{c}(\sigma, inherit\ B\ R) \in \Gamma \\ (B.\rho, \texttt{;IFL;}\ (l)\ \phi\ \texttt{;IFL;}) \in \Gamma \\ \texttt{;IFL;}(l' : \rho.l)\ \phi'\texttt{;IFL;} \in R\end{array}}{add(\sigma.\rho, \texttt{;IFL;}(l')\ \phi' \sqcap \phi\ \texttt{;IFL;})} \quad \text{(N-2")}\ \frac{\begin{array}{c}(\sigma, inherit\ B\ R) \in \Gamma \\ (B.\rho, \texttt{;IFL;}\ (l)\ \phi\ \texttt{;IFL;}) \in \Gamma \\ \neg\exists l', \phi' : \texttt{;IFL;}(l' : \rho.l)\ \phi'\texttt{;IFL;} \in R\end{array}}{add(\sigma.\rho, \texttt{;IFL;}(l')\ \phi\ \texttt{;IFL;})}$$

$$\text{(N-3)}\ \frac{\overline{eval}_{\sigma;\#}(m) = m'}{(\sigma, call\ m([a])\ R) \to (\sigma, call\ m'([a])\ R)} \quad \text{(N-4)}\ \frac{\begin{array}{c}(\sigma, call\ m([a])\ R) \in \Gamma \\ (m, d) \in \Gamma\end{array}}{add(\sigma, d)}$$

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

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

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

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

Figure B.4: IFCIL normalization rules.

B.3 Syntax and semantics of IFCIL

The grammar for IFCIL is obtained by updating the two rules of *command* concerning *call* and *blockinherit* in the following way.

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

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 B.4, 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 ($1 \leq i \leq 6$).

The semantics $(G, \mathbb{R})$ of a IFCIL configuration Σ is obtained by applying the rules in Figure B.2 for G, and the following one for $\mathbb{R}$.

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

B.4 Verification: Proofs

In the following we will consider the redefinition of the grammar of P in Section 3.4, and the following definition of $\triangleright_I$, which is trivially equivalent to the original one.

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

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

Lemma B.2. Let π be a path in the information diagram I with K its KTS, and let P be a information flow kind. Then

$$(1) \qquad \pi \triangleright_I P \Rightarrow \exists w \in (\!|\pi|\!) : w \models (\!|P|\!)$$
$$(2) \qquad w \models (\!|P|\!) \Rightarrow \exists \pi : w \in (\!|\pi|\!) \ \wedge \ \pi \triangleright_I P.$$

Proof. We proceed by induction on P. For P of the form $\mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}'$ or $\mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}'$ the properties trivially hold. For the last two cases of P, we separately prove (1) and (2).

Case $P = \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}'\ P'$. Consider (1) and assume $\pi \triangleright_I P$. By the definition of $\triangleright_I$, $\pi = (n, o, n')\ \pi'$, with $o \cap o' \neq \emptyset$. We also know by the induction hypothesis that there exists a $w' \in (\!|\pi'|\!)$ such that $w' \models (\!|P'|\!)$. Since $o \cap o' \neq \emptyset$, there exists an $op \in o \cap o'$. Take $w = (n, op, n')\ w'$. Then $w \in (\!|\pi|\!)$ holds by the definition, and also $w \models (\!|P|\!)$.

Consider now (2), and assume $w \models (\!|P|\!)$. Since $(\!|P|\!) = m \wedge \bigvee_{op \in o'}(op) \ \wedge X((\!|P|\!))$, $w = (n, op, n')w'$ with $n \in ta(m)$ and $op \in o'$ trivially follows from the semantics of LTL. We also know by the induction hypothesis that there exists π' such that $\pi' \in (\!|w'|\!)$ and $\pi' \triangleright_I P'$. Take $\pi = (n, \{op\}, n')\pi'$. Then $w \in (\!|\pi|\!)$ holds by the definition, and also $\pi \triangleright_I P$.

Case $P = \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}'\ P'$. Consider (1) and assume $\pi \triangleright_I P$. We separately consider the two disjunctive cases of the definition of $\triangleright$ for this case. If $(\mathtt{n}, o, \mathtt{n}')\triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>} * \wedge \pi \triangleright_I P)$, then the proof is similar to the one for $P = \mathtt{m}\ \mathtt{[}o'\mathtt{]>}\ \mathtt{m}'\ P'$. Otherwise, by the definition of $\triangleright_I$ we have that $\pi = (n, o, n')\ \pi'$ with $(n, o, n')\triangleright_I m + [o'] > *$ and $\pi' \triangleright_I * + [o'] > *\ P'$. By the definition $ocapo' \neq \emptyset$ and thus take $op \in ocapo'$. By the definition, $\pi' \triangleright_I * + [o'] > *\ P'$ implies that $\pi' = \pi''\ \pi'''$ such that $\pi'' \triangleright_I * + [o'] > *$ and $\pi''' \triangleright P'$. Let $\pi'' = (n_1, o_1, n_2) \ldots (n_{p-1}, o_{p-1}, n_p)$, by the definition of $\triangleright_I$, for each i there exists an $op_i \in o' \cap o_i$. Since $\pi''' \triangleright P'$, by the induction hypothesis, there exists a w' such that $w \in (\!|\pi'''|\!)$ and $w''' \models P'$. Then the thesis trivially holds for $w = (n, op, n')(n_1, op_1, n_2) \ldots (n_{p-1}, op_{p-1}, n_p)w'$.

Consider now (2) and assume $w \models (\!|P|\!)$. We have that $w \models \mathtt{m} \wedge \bigvee_{\mathtt{op} \in \mathtt{o}'}(\mathtt{op}) \wedge X(\bigvee_{\mathtt{op} \in \mathtt{o}'}(\mathtt{op})\ U\ (\!|P'|\!))$. By the semantics of LTL, this implies that $w = (n, op, n')(n_1, op_1, n_2) \ldots (n_{p-1}, op_{p-1}, n_p)w'$, with $w' \models (\!|P'|\!)$, $op \in o'$, and $\forall i : op_i \in o'$. By the induction hypothesis, there exists a π' such that $w' \in (\!|\pi'|\!)$ and $\pi' \triangleright_I P'$. Take $\pi = (n, \{op\}, n')(n_1, \{op_1\}, n_2) \ldots (n_{p-1}, \{op_{p-1}\}, n_p)\pi'$. Then $w \in (\!|\pi|\!)$ trivially holds by the definition. Finally, if $p = 0$, then the first case in the definition of $\triangleright_I$ for $\pi \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>}\ \mathtt{m}'\ P'$ holds; otherwise the second case holds since $(\mathtt{n}, o, \mathtt{n}') \triangleright_I \mathtt{m}\ \mathtt{+[}o'\mathtt{]>} *$, $(n_1, \{op_1\}, n_2) \ldots (n_{p-1}, \{op_{p-1}\}, n_p) \triangleright_I *\mathtt{+[}o'\mathtt{]>}*$, and $\pi' \triangleright_I P'$.

□

Lemma B.3. Let I be an information diagram with K its KTS, then $W = \bigcup_{\pi \text{ in } I} (\!|\pi|\!)$.

Proof. Trivially follows from the definition of K and $(\!|\pi|\!)$. □

Theorem 3.2 (Correctness)**.** 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

$$K \vdash \mathbb{R} \;\Rightarrow\; I \models \mathbb{R}.$$

Proof. We prove $\forall \mathcal{R} : K \vdash \mathcal{R} \Rightarrow I \models \mathcal{R}$, considering all possible forms for $\mathcal{R}$, and assuming $K \vdash \mathcal{R}$.

If $\mathcal{R} = P$, $\neg \forall w \in W : w \models \neg(\!(P)\!)$. Equivalently, there exists a $w \in W$ such that $w \models (\!(P)\!)$. By Lemma B.2, there exist a path π in I such that $w \in (\!(\pi)\!)$ and $\pi \triangleright_I P$, hence $I \models \mathcal{R}$.

If $\mathcal{R} = \check{} P$, $\forall w \in W : w \models \neg(\!(P)\!)$. Assume by refutation that $\neg I \models \mathcal{R}$, then $I \models P$, i.e., there exists a π in I such that $\pi \triangleright_I P$. But then, from Lemma B.2, we know that exists a $\bar{w} \in (\!(\pi)\!) \subseteq W$ such that $\bar{w} \models (\!(P)\!)$, and thus $\bar{w} \models \neg(\!(P)\!)$ does not hold.

If $\mathcal{R} = P : P'$, $\forall w \in W : w \models \neg(\!(P)\!) \vee w \models (\!(P \sqcap P')\!)$. By Lemma B.2 and B.3, this implies that $\forall \pi$ in I either $\pi \triangleright_I P$ does not hold or $\pi \triangleright_I P \sqcap P'$. By Lemma B.1 and definition of $\sqcap$, the latter implies that $\pi \triangleright_I P'$, and thus the thesis follows. $\square$

Appendix C

Technical Details and Proofs of Chapter 4

C.1 CLNL* Immersion: Proofs

In the following we use $\vdash_{CLNL}$ and $\vdash_{CLNL*}$ for computational sequents in CLNL and CNLN* respectively. Before proving the correctness of CLNL* deduction rules, we give the following straightforward lemma.

Lemma C.1. For all Ψ, φ and φ', if $\varphi \subseteq \varphi'$ then, φ'' and a CLNL proof $\Pi_{\varphi,\varphi'}$ exist such that

$$\frac{\Pi_{\varphi,\varphi'}}{\Psi; \varphi' \vdash \varphi \otimes \varphi''}$$

Proof. By trivial induction on the cardinality of φ' using (L-Ax), ($\otimes$-left), ($\otimes$-right). □

For convenience, we also enrich CLNL with the extra rule (LL-cut$'$), defined as

$$\frac{\Psi; \Phi \vdash \varphi \quad \Psi; \Phi', \varphi \vdash \varphi'}{\Psi; \Phi, \Phi' \vdash \varphi'}\,(\text{LL-cut}')$$

This rule is clearly admissible in CLNL, every occurrence can be substituted by

$$\cfrac{\cfrac{\Psi; \Phi \vdash \varphi \quad \Psi; \Phi', \varphi \vdash \varphi'}{\Psi, \Psi; \Phi, \Phi' \vdash \varphi'}\,(\text{LL-cut})}{\Psi; \Phi, \Phi' \vdash \varphi'}\,(\text{L-Cont})$$

We also enrich CNLN* with the extra rule ($-\!\circ\!\infty$-Merge'), that generalizes

(⊸-Merge), and is clearly derivable by multiple applications of (⊸-Merge).

$$\frac{\Omega;\Theta,\left\{\bigotimes_{j=1}^{n_i}\delta_{i,j}\multimap\bigotimes_{j=1}^{n_i}\delta'_{i,j}\;\middle|\;i\in[1,k]\right\},\Delta,\mathbb{S}\vdash S}{\Omega;\Theta,\{\delta_{i,j}\multimap\delta'_{i,j}\mid i\in[1,k],j\in[1,n_i]\},\Delta,\mathbb{S}\vdash S}\,(\multimap\text{-Merge'})$$

We give the following auxiliary lemma.

Lemma C.2. If $\bigotimes_{i=1}^{n}\delta'_i\subseteq\bigotimes_{i=1}^{n}\delta_i$ then, a LNLC proof exists for

$$\{\delta_i\multimap\delta'_i\mid i\in[1,n]\}\vdash\bigotimes_{i=1}^{n}\delta_i$$

Proof. The LNLC proof is

$$\frac{\dfrac{\Pi_{\subseteq 1}}{\bigotimes_{i=1}^{n}\delta_i\vdash\delta_1\otimes\varphi}\qquad\Pi_{1,n}}{\{\{\delta_i\multimap\delta'_i\mid i\in[1,n]\}\vdash\bigotimes_{i=1}^{n}\delta_i}\,(\multimap\text{-fix})$$

Where $\Pi_{j,n}$ is defined recursively as

$$\frac{\Pi_0}{\{\delta_i\mid i\in[1,n]\}\vdash\bigotimes_{i=1}^{n}\delta_i}\,(\multimap\text{-fix})$$

if $j=n$, and otherwise as

$$\frac{\dfrac{\Pi_{\subseteq j+1}}{\bigotimes_{i=1}^{n}\delta_i\vdash\delta_{j+1}\otimes\varphi}\qquad\Pi_{j+1,n}}{\{\delta_i\mid i\in[1,j]\},\{\delta_i\multimap\delta'_i\mid i\in[j+1,n]\}\vdash\bigotimes_{i=1}^{n}\delta_i}\,(\multimap\text{-fix})$$

for suitable Π_0 and $\Pi_{\subseteq j}$ for $j\in[1,n]$.

The existence of $\Pi_{\subseteq j}$ trivially follows from Lemma C.1. The proof Π_0 is simply obtained by applying (L-Ax) and (⊗-right). □

We now prove the main result.

Theorem 4.1 (CLNL* immersion)**.** For all Ω, Θ, Δ, $\mathbb{S}$ and S,

$$\Omega;\Theta,\Delta,\mathbb{S}\vdash S$$

is valid in CLNL* only if it is valid in CLNL.

Proof. We prove that If a proof Π_{CLNL*} exists in CLNL* for a computational sequent, then a proof Π_{CLNL} exists in CNLN for the same sequent.

The only rules of Π_{CLNL*} that are not shared by CLNL are ($\multimap\!\!\infty$-Spend), and ($\multimap\!\!\infty$-Merge'), thus we have only to prove that their occurrences can be substituted with CLNL derivations. We prove it by induction on the length of Π_{CLNL*}. The case of depth 1 is trivial, no proof can contain $\multimap\!\!\infty$ rules since the only two rules without premises are (I-right) and (L-Ax).

Assume the thesis holds for any proof of depth m, and take Π of length $m+1$. We reduce to the case in which ($\multimap\!\!\infty$-Spend) or ($\multimap\!\!\infty$-Merge) is the last rule of Π_{CLNL*}, in all other cases the thesis trivially follows from induction hypothesis.

Case ($\multimap\!\!\infty$-Spend): Assume the last rule of Π_{CLNL*} to be ($\multimap\!\!\infty$-Spend), then Π_{CLNL*} is

$$\dfrac{\delta \subseteq \delta' \quad \dfrac{\Pi'_{CLNL*}}{\Omega; \Theta, \Delta, \delta', \mathbb{S} \vdash S}}{\Omega; \Theta, \delta \multimap\!\!\infty \delta', \Delta, \mathbb{S} \vdash S}\,(\multimap\!\!\infty\text{-Spend})$$

for some Π'_{CLNL*} of depth m. By induction hypothesis on Π'_{CLNL*}, and by Lemma C.1, exist Π'_{CLNL} and $\Pi_{\delta,\delta'}$ such that

$$\dfrac{\dfrac{\dfrac{\Pi_{\delta,\delta'}}{\delta' \vdash \delta \otimes \varphi} \quad \dfrac{}{\delta' \vdash \delta'}\,(\text{L-Ax})}{\delta \multimap\!\!\infty \delta' \vdash \delta'}\,(\multimap\!\!\infty\text{-fix}) \quad \dfrac{\Pi'_{CLNL}}{\Omega; \Theta, \Delta, \delta', \mathbb{S} \vdash S}}{\Omega; \Theta, \delta \multimap\!\!\infty \delta', \Delta, \mathbb{S} \vdash S}\,(\text{LL-cut})$$

Case ($\multimap\!\!\infty$-Merge'): Finally, assume the last rule of Π_{CLNL*} to be ($\multimap\!\!\infty$-Merge'), then Π_{CLNL*} is

$$\dfrac{\dfrac{\Pi'_{CLNL*}}{\Omega; \Theta, \left\{ \bigotimes_{j=1}^{n_i} \delta_{i,j} \multimap\!\!\infty \bigotimes_{j=1}^{n_i} \delta'_{i,j} \;\middle|\; i \in [1,k] \right\}, \Delta, \mathbb{S} \vdash S}}{\Omega; \Theta, \{\delta_{i,j} \multimap\!\!\infty \delta'_{i,j} \mid i \in [1,k], j \in [1,n_i]\}, \Delta, \mathbb{S} \vdash S}\,(\multimap\!\!\infty\text{-Merge'})$$

for some Π'_{CLNL*} with depth m.

We proceed by cases on the last rule (r) of Π'_{CLNL*}.

Case (r) = (L-Cont), (L-Weak), (L-$\wedge$-left1), (L-$\wedge$-left2), (G-left), ($\otimes$-left-Θ), ($\otimes$-left-Δ) or ($\otimes$-left-$\mathbb{S}$): The property trivially holds for the following rules: (L-Cont), (L-Weak), (L-$\wedge$-left1), (L-$\wedge$-left2), (G-left), ($\otimes$-left-Θ), ($\otimes$-left-Δ), ($\otimes$-left-$\mathbb{S}$). Indeed, when considering them bottom-up, none of them removes or modifies Θ, thus, $\delta \otimes \delta'' \multimap\!\!\infty \delta' \otimes \delta'''$ is unchanged in the

premises of these rules. Formally, for any of these rules, the proof Π_{CLNL*} is

$$\dfrac{\dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega';\Theta',\Theta'_{Merge},\Delta',\mathbb{S}' \vdash S}\,(\mathrm{r})}{\Omega;\Theta,\Theta'_{Merge},\Delta,\mathbb{S} \vdash S}\,(\multimap\!\!\infty\text{-Merge'})}{\Omega;\Theta,\Theta_{Merge},\Delta,\mathbb{S} \vdash S}$$

for some Π''_{CLNL*} of depth $m-1$ and for suitable Θ_{Merge} and Θ'_{Merge}. This proof can be rewritten as

$$\dfrac{\dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega';\Theta',\Theta'_{Merge},\Delta',\mathbb{S}' \vdash S}\,(\multimap\!\!\infty\text{-Merge'})}{\Omega';\Theta',\Theta_{Merge},\Delta',\mathbb{S}' \vdash S}\,(\mathrm{r})}{\Omega;\Theta,\Theta_{Merge},\Delta,\mathbb{S} \vdash S}$$

By induction hypothesis on the proof of length m

$$\dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega';\Theta',\Theta'_{Merge},\Delta',\mathbb{S}' \vdash S}\,(\multimap\!\!\infty\text{-Merge'})}{\Omega';\Theta',\Theta_{Merge},\Delta',\mathbb{S}' \vdash S}$$

we know that a proof Π'_{CLNL} exists in CLNL such that

$$\dfrac{\dfrac{\Pi'_{CLNL}}{\Omega';\Theta',\Theta_{Merge},\Delta',\mathbb{S}' \vdash S}\,(\mathrm{r})}{\Omega;\Theta,\Theta_{Merge},\Delta,\mathbb{S} \vdash S}$$

Case (r) = (L-$\to$-left), ($\multimap$-left), ($\otimes$-right): A similar procedure applies to these rules, with the only difference that they have two premises.

Case (r) = ($\multimap\!\!\infty$-Spend): If the last rule of Π'_{CLNL*} is ($\multimap\!\!\infty$-Spend), the proof Π_{CLNL*} is

$$\dfrac{\dfrac{\delta \subseteq \delta' \qquad \dfrac{\Pi''_{CLNL*}}{\Omega;\Theta',\Delta,\delta',\mathbb{S} \vdash S}}{\Omega;\Theta,\left\{\bigotimes_{j=1}^{n_i}\delta_{i,j} \multimap\!\!\infty \bigotimes_{j=1}^{n_i}\delta'_{i,j} \;\middle|\; i \in [1,k]\right\},\Delta,\mathbb{S} \vdash S}\,(\multimap\!\!\infty\text{-Spend})}{\Omega;\Theta,\{\delta_{i,j} \multimap\!\!\infty \delta'_{i,j} \mid i \in [1,k], j \in [1,n_i]\},\Delta,\mathbb{S} \vdash S}\,(\multimap\!\!\infty\text{-Merge'})$$

for some suitable Θ', δ, δ', and Π''_{CLNL*} of depth $m-1$. In the following let Θ_{Merge} and Θ'_{Merge} be

$$\{\delta_{i,j} \multimap\!\!\infty \delta'_{i,j} \mid i \in [1,k], j \in [1,n_i]\} \quad \text{and} \quad \left\{\bigotimes_{j=1}^{n_i}\delta_{i,j} \multimap\!\!\infty \bigotimes_{j=1}^{n_i}\delta'_{i,j} \;\middle|\; i \in [1,k]\right\}$$

respectively. We consider two cases, either $\delta \multimap\!\!\circ\, \delta'$ is in Θ, or it is in Θ'_{Merge}. Assume $\delta \multimap\!\!\circ\, \delta'$ is in Θ, then $\Theta' = (\Theta \setminus \{\delta \multimap\!\!\circ\, \delta'\}), \Theta'_{Merge}$. Thus we can rewrite the proof in the following form and reduce to a previous case.

$$\dfrac{\delta \subseteq \delta' \quad \dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega; \Theta', \Delta, \delta', \mathbb{S} \vdash S}}{\Omega; \Theta \setminus \{\delta \multimap\!\!\circ\, \delta'\}, \Theta_{Merge}, \Delta, \delta', \mathbb{S} \vdash S}\,(\multimap\!\!\circ\text{-Merge'})}{\Omega; \Theta, \Theta_{Merge}, \Delta, \mathbb{S} \vdash S}\,(\multimap\!\!\circ\text{-Spend})$$

Otherwise, $\delta \multimap\!\!\circ\, \delta'$ is in Θ'_{Merge}, i.e.,

$$\delta = \bigotimes_{j=1}^{n_q} \delta_{q,j} \qquad \text{and} \qquad \delta' = \bigotimes_{j=1}^{n_q} \delta'_{q,j}$$

for some $q \in [1, k]$. Then $\Theta' = \Theta, (\Theta'_{Merge} \setminus \{\delta \multimap\!\!\circ\, \delta'\})$, and the proof can be rewritten as follows.

$$\dfrac{\dfrac{\delta \subseteq \delta' \quad \dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega; \Theta', \Delta, \delta', \mathbb{S} \vdash S}}{\Omega; \Theta, \{\delta_{i,j} \multimap\!\!\circ\, \delta'_{i,j} \mid i \neq q \in [1,k], j \in [1, n_i]\}, \Delta, \delta', \mathbb{S} \vdash S}\,(\multimap\!\!\circ\text{-Merge'})}{\Omega; \Theta, \delta \multimap\!\!\circ\, \delta', \{\delta_{i,j} \multimap\!\!\circ\, \delta'_{i,j} \mid i \neq q \in [1,k], j \in [1, n_i]\}, \Delta, \mathbb{S} \vdash S}\,(\multimap\!\!\circ\text{-Spend})}{\Omega; \Theta, \{\delta_{i,j} \multimap\!\!\circ\, \delta'_{i,j} \mid i \in [1,k], j \in [1, n_i]\}, \Delta, \mathbb{S} \vdash S}\,(\multimap\!\!\circ\text{-Merge'})$$

By induction hypothesis on the proof of length m

$$\dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega; \Theta', \Delta, \delta', \mathbb{S} \vdash S}}{\Omega; \Theta, \{\delta_{i,j} \multimap\!\!\circ\, \delta'_{i,j} \mid i \neq q \in [1,k], j \in [1, n_i]\}, \Delta, \delta', \mathbb{S} \vdash S}\,(\multimap\!\!\circ\text{-Merge'})$$

a proof $\Pi_{CLNL-right}$ exists in CLNL for

$$\Omega; \Theta, \{\delta_{i,j} \multimap\!\!\circ\, \delta'_{i,j} \mid i \neq q \in [1,k], j \in [1, n_i]\}, \Delta, \delta', \mathbb{S} \vdash S$$

By Lemma C.2, a proof $\Pi_{CLNL-left}$ exists in LNLC for

$$\{\delta_{q,j} \multimap\!\!\circ\, \delta'_{q,j} \mid j \in [1, n_i]\} \vdash \bigotimes_{j=1}^{n} \delta'_{q,j}$$

Then, a CLNL proof exists

$$\dfrac{\Pi_{CLNL-left} \qquad\qquad \Pi_{CLNL-right}}{\Omega; \Theta, \{\delta_{i,j} \multimap\!\!\circ\, \delta'_{i,j} \mid i \in [1,k], j \in [1, n_i]\}, \Delta, \mathbb{S} \vdash S}\,(\text{LL-cut})$$

Case (r) = (⊸∘-Merge'): Finally, let the last rule of Π'_{CLNL*} be (⊸ ⊸∘-Merge'). Without loosing generality we assume the two (⊸∘-Merge') rules

applies on the same set (note that n_i may be 1 for some i). Then the proof is as follows.

$$\dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega;\Theta,\left\{\bigotimes_{i=1}^{n_z}\bigotimes_{j=1}^{n_{z,i}}\delta_{z,i,j} \multimap\!\!\multimap \bigotimes_{i=1}^{n_z}\bigotimes_{j=1}^{n_{z,i}}\delta'_{z,i,j} \;\middle|\; z\in[1,k]\right\},\Delta,\mathbb{S}\vdash S}(\multimap\!\!\multimap\text{-Merge'})}{\Omega;\Theta,\left\{\bigotimes_{j=1}^{n_{z,i}}\delta_{z,i,j} \multimap\!\!\multimap \bigotimes_{j=1}^{n_{z,i}}\delta'_{z,i,j} \;\middle|\; z\in[1,k], i\in[1,n_z]\right\},\Delta,\mathbb{S}\vdash S}(\multimap\!\!\multimap\text{-Merge'})$$

$$\Omega;\Theta,\{\delta_{z,i,j}\multimap\!\!\multimap\delta'_{z,i,j}\mid z\in[1,k],i\in[1,n_z]j\in[1,n_{z,i}]\},\Delta,\mathbb{S}\vdash S$$

And it can be rewritten in the following way as a proof of length m.

$$\dfrac{\dfrac{\Pi''_{CLNL*}}{\Omega;\Theta,\left\{\bigotimes_{(i,j)=(1,i)}^{(n_z,n_{z,i})}\delta_{z,i,j} \multimap\!\!\multimap \bigotimes_{(i,j)=(1,i)}^{(n_z,n_{z,i})}\delta'_{z,i,j} \;\middle|\; z\in[1,k]\right\},\Delta,\mathbb{S}\vdash S}(\multimap\!\!\multimap\text{-Merge'})}{\Omega;\Theta,\{\delta_{z,i,j}\multimap\!\!\multimap\delta'_{z,i,j}\mid z\in[1,k],i\in[1,n_z]j\in[1,n_{z,i}]\},\Delta,\mathbb{S}\vdash S}(\multimap\!\!\multimap\text{-Merge'})$$

Thus the thesis fallows by induction hypothesis. □

C.2 CLNL* Decidability: Proofs

We start by reducing to decide normalized proofs only. Then we prove that our algorithm is correct.

C.2.1 Normalized CLNL* proofs

We will prove the normalization of CLNL* proofs by mapping them to proofs in a derived logic, called CLNL*2. Then we will reorder proofs in CLNL*2 and map them back to CLNL* where a final reordering takes place.

Let CLNL*2 be the logic defined exactly as CLNL* but for the rules ($\multimap$-left) and ($\otimes$-right) that are substituted with the following.

$$\dfrac{\Omega;\Theta,\Delta,\mathbb{S}\vdash S \qquad \Omega';\Theta',\Delta',\mathbb{S}',S'\vdash S''}{\Omega,\Omega';\Theta,\Theta',\Delta,\Delta',\mathbb{S},\mathbb{S}',S\multimap S'\vdash S''}(\multimap\text{-left'})$$

$$\dfrac{\Omega;\Theta,\Delta,\mathbb{S}\vdash S \qquad \Omega';\Theta',\Delta',\mathbb{S}'\vdash S'}{\Omega,\Omega';\Theta,\Theta',\Delta,\Delta',\mathbb{S},\mathbb{S}'\vdash S\otimes S'}(\otimes\text{-right'})$$

Lemma C.3. ($\multimap$-left') is a derivable in CLNL* as

$$\dfrac{\dfrac{\Omega;\Theta,\Delta,\mathbb{S}\vdash S}{\Omega,\Omega';\Theta,\Delta,\mathbb{S}\vdash S}(\text{L-Weak}) \quad \dfrac{\Omega';\Theta',\Delta',\mathbb{S}',S'\vdash S''}{\Omega,\Omega';\Theta',\Delta',\mathbb{S}',S'\vdash S''}(\text{L-Weak})}{\Omega,\Omega';\Theta,\Theta',\Delta,\Delta',\mathbb{S},\mathbb{S}',S\multimap S'\vdash S''}(\multimap\text{-left})$$

and (⊗-right') is a derivable in CLNL* as

$$\dfrac{\dfrac{\Omega;\Theta,\Delta,\mathbb{S}\vdash S}{\Omega,\Omega';\Theta,\Delta,\mathbb{S}\vdash S}\,\text{(L-Weak)}\quad\dfrac{\Omega';\Theta',\Delta',\mathbb{S}'\vdash S'}{\Omega,\Omega';\Theta',\Delta',\mathbb{S}'\vdash S'}\,\text{(L-Weak)}}{\Omega,\Omega';\Theta,\Theta',\Delta,\Delta',\mathbb{S},\mathbb{S}'\vdash S\otimes S'}\,(\otimes\text{-right})$$

Proof. Holds by definition. □

From Lemma C.3 we can immediately derive the following.

Corollary C.1. If a CLNL* proof exists for a sequent, then an equivalent one in CLNL*2 exists.

Proof. By substituting non common rules with their definitions in Lemma C.3. □

Lemma C.4. If an instance of (⊗-right') is such that $\Omega,\Omega'=\emptyset$, then it is also an instance of (⊗-right). If an instance of (⊸-left') is such that $\Omega,\Omega'=\emptyset$, then it is also an instance of (⊸-left).

Proof. By definition. □

We gives the following definitions and auxiliary lemmas about reordering rules in CLNL*2.

Recall that

$$\begin{aligned}
Sr &= \{\text{(L-Weak), (L-Cont)}\}\\
Cr &= \{\text{(C-}\top\text{), (C-Ax), (C-Cont), (C-Weak), (C-}\wedge\text{-left1), (C-}\wedge\text{-left2),}\\
&\quad\text{(C-}\multimap\text{-left), (C-}\multimap\text{-right), (L-}\wedge\text{-right),(L-}\wedge\text{-left2),(L-}\multimap\text{-left), (CL-Cut)}\}\\
Lr &= \{\text{(}\multimap\text{-left),(}\otimes\text{-right),(}\otimes\text{-left-}\Theta\text{),(}\otimes\text{-left-}\Delta\text{),(}\otimes\text{-left-}\mathbb{S}\text{)}\}\\
Gr &= \{\text{(G-left-}\theta\text{), (G-left-}\delta\text{)}\}\\
Pr &= \{\text{(}\multimap\!\!\infty\text{-Spend), (}\multimap\!\!\infty\text{-Merge)}\}
\end{aligned}$$

Let Lr' be the set of CLNL*2 rules defined as follows.

$$Lr' = \{\text{(}\multimap\text{-left'),(}\otimes\text{-right'),(}\otimes\text{-left-}\Theta\text{),(}\otimes\text{-left-}\Delta\text{),(}\otimes\text{-left-}\mathbb{S}\text{)}\}$$

In the following, we call *unitary* the rules with one premise only.

Lemma C.5. A derivation with two rules $r\in Sr\cup Cr\cup Gr$ followed by $r'\in Lr'\cup Gr\cup Pr$ can be rewritten as an equivalent derivation where no rule $Lr'\cup Gr\cup Pr$ follows a rule in $Sr\cup Cr\cup Gr$.

Proof. The only non trivial cases are $r=$ (L-$\to$-left) or (CL-Cut), and $r'=$ (⊸-left') or (⊗-right'). Take $r=$ (L-$\to$-left) and $r'=$ (⊗-right'), and let r be applied to the left derivation

$$\dfrac{\dfrac{\Omega\Vdash\omega\quad\Omega',\omega';\Theta,\Delta,\mathbb{S}\vdash S}{\Omega,\Omega',\omega\to\omega';\Theta,\Delta,\mathbb{S}\vdash S}\,\text{(L-}\to\text{-left)}\qquad\Omega'';\Theta',\Delta',\mathbb{S}'\vdash S'}{\Omega,\Omega',\Omega'',\omega\to\omega';\Theta,\Theta',\Delta,\Delta',\mathbb{S},\mathbb{S}'\vdash S\otimes S'}\,(\otimes\text{-right'})$$

We can swap the rules as follows.

$$\dfrac{\Omega \Vdash \omega \quad \dfrac{\Omega'';\Theta',\Delta',\mathbb{S}' \vdash S' \quad \Omega',\omega';\Theta,\Delta,\mathbb{S} \vdash S}{\Omega',\Omega'',\omega';\Theta,\Theta',\Delta,\Delta',\mathbb{S},\mathbb{S}' \vdash S \otimes S'}\,(\otimes\text{-right'})}{\Omega,\Omega',\Omega'',\omega \to \omega';\Theta,\Theta',\Delta,\Delta',\mathbb{S},\mathbb{S}' \vdash S \otimes S'}\,(\text{L-}\to\text{-left})$$

If r is (CL-Cut) or it is applied to the right derivation the proof is almost identical.

Take r = (L-$\to$-left) and r' = ($\multimap$-left'), and let r be applied to the left derivation

$$\dfrac{\dfrac{\Omega \Vdash \omega \quad \Omega',\omega';\Theta,\Delta,\mathbb{S} \vdash S}{\Omega,\Omega',\omega \to \omega';\Theta,\Delta,\mathbb{S} \vdash S}\,(\text{L-}\to\text{-left}) \qquad \Omega'';\Theta',\Delta',\mathbb{S}',S' \vdash S''}{\Omega,\Omega',\Omega'',\omega \to \omega';\Theta,\Theta',\Delta,\Delta',S \multimap S',\mathbb{S},\mathbb{S}' \vdash S''}\,(\multimap\text{-left'})$$

We can swap the rules as follows.

$$\dfrac{\Omega \Vdash \omega \quad \dfrac{\Omega',\omega';\Theta,\Delta,\mathbb{S} \vdash S \quad \Omega'';\Theta',\Delta',\mathbb{S}',S' \vdash S''}{\Omega',\Omega'',\omega';\Theta,\Theta',\Delta,\Delta',S \multimap S',\mathbb{S},\mathbb{S}' \vdash S''}\,(\multimap\text{-left'})}{\Omega,\Omega',\Omega'',\omega \to \omega';\Theta,\Theta',\Delta,\Delta',S \multimap S',\mathbb{S},\mathbb{S}' \vdash S''}\,(\text{L-}\to\text{-left})$$

If r is (CL-Cut) or it is applied to the right derivation the proof is almost identical. □

Lemma C.6. If derivation Π with two rules $r \in Sr \cup Cr$ followed by $r' \in Gr$ exists in CLNL*2, then an equivalent derivation Π' exists where no rule in Gr follows a rule in $Sr \cup Cr$.

Proof. Let r and r' be (L-$\to$-left) and (G-left-θ) respectively, i.e., let Π be

$$\dfrac{\dfrac{\Omega \Vdash \omega \quad \Omega',\omega';\Theta,\theta,\Delta,\mathbb{S} \vdash S}{\Omega,\Omega',\omega \to \omega';\Theta,\theta,\Delta,\mathbb{S},\vdash S}\,(\text{L-}\to\text{-left})}{\Omega,\Omega',\omega \to \omega',G(\theta);\Theta,\Delta,\mathbb{S},\vdash S}\,(\text{G-left-}\theta)$$

Then, Π' is as follows.

$$\dfrac{\Omega \Vdash \omega \quad \dfrac{\Omega',\omega';\Theta,\theta,\Delta,\mathbb{S} \vdash S}{\Omega',\omega',G(\theta);\Theta,\Delta,\mathbb{S},\vdash S}\,(\text{G-left-}\theta)}{\Omega,\Omega',\omega \to \omega',G(\theta);\Theta,\Delta,\mathbb{S},\vdash S}\,(\text{L-}\to\text{-left})$$

The proof is almost identical for any $r \in Cr$ and $r' \in Gr$.

Let r and r' be (L-Weak) and (G-left-θ) respectively, i.e., let Π be

$$\dfrac{\dfrac{\Omega;\Theta,\theta,\Delta,\mathbb{S} \vdash S}{\Omega,\omega;\Theta,\theta,\Delta,\mathbb{S},\vdash S}\,(\text{L-Weak})}{\Omega,\omega,G(\theta);\Theta,\Delta,\mathbb{S},\vdash S}\,(\text{G-left-}\theta)$$

Then, Π' is as follows.

$$\dfrac{\dfrac{\Omega; \Theta, \theta, \Delta, \mathbb{S} \vdash S}{\Omega, G(\theta); \Theta, \Delta, \mathbb{S}, \vdash S}\,(\text{G-left-}\theta)}{\Omega, \omega, G(\theta); \Theta, \Delta, \mathbb{S}, \vdash S}\,(\text{L-Weak})$$

The proof is almost identical for (G-left-δ).

Let r and r' be (L-Cont) and (G-left-θ) respectively, i.e., let Π be

$$\dfrac{\dfrac{\Omega, \omega, \omega; \Theta, \theta, \Delta, \mathbb{S} \vdash S}{\Omega, \omega; \Theta, \theta, \Delta, \mathbb{S}, \vdash S}\,(\text{L-Cont})}{\Omega, \omega, G(\theta); \Theta, \Delta, \mathbb{S}, \vdash S}\,(\text{G-left-}\theta)$$

Then, Π' is as follows.

$$\dfrac{\dfrac{\Omega, \omega, \omega; \Theta, \theta, \Delta, \mathbb{S} \vdash S}{\Omega, \omega, \omega, G(\theta); \Theta, \Delta, \mathbb{S}, \vdash S}\,(\text{G-left-}\theta)}{\Omega, \omega, G(\theta); \Theta, \Delta, \mathbb{S}, \vdash S}\,(\text{L-Cont})$$

The proof is almost identical for (G-left-δ). □

Lemma C.7. If derivation Π with two rules $r \in Pr$ followed by $r' \in Lr'$ exists in CLNL*2, then an equivalent derivation Π' exists where no rule in Lr' follows a rule in Pr.

Proof. The only non trivial is $r' =$ ($\multimap$-left') or ($\otimes$-right').

Take $r' =$ ($\otimes$-right'), and let $r =$ ($\multimap\!\!\otimes$-Merge) be applied to the left derivation

$$\dfrac{\dfrac{\Omega; \Theta, \delta \otimes \delta'' \multimap\!\!\otimes\, \delta' \otimes \delta''', \theta, \Delta, \mathbb{S} \vdash S}{\Omega; \Theta, \delta \multimap\!\!\otimes\, \delta', \delta'' \multimap\!\!\otimes\, \delta''', \Delta, \mathbb{S} \vdash S}\,(\multimap\!\!\otimes\text{-Merge}) \qquad \Omega'; \Theta', \Delta', \mathbb{S}' \vdash S'}{\Omega, \Omega'; \Theta, \Theta', \delta \multimap\!\!\otimes\, \delta', \delta'' \multimap\!\!\otimes\, \delta''', \Delta, \Delta', \mathbb{S}, \mathbb{S}' \vdash S \otimes S'}\,(\otimes\text{-right'})$$

We can swap the rules as follows.

$$\dfrac{\dfrac{\Omega; \Theta, \delta \otimes \delta'' \multimap\!\!\otimes\, \delta' \otimes \delta''', \Delta, \mathbb{S} \vdash S \qquad \Omega'; \Theta', \Delta', \mathbb{S}' \vdash S'}{\Omega, \Omega'; \Theta, \Theta', \delta \otimes \delta'' \multimap\!\!\otimes\, \delta' \otimes \delta''', \Delta, \Delta', \mathbb{S}, \mathbb{S}' \vdash S \otimes S'}\,(\otimes\text{-right'})}{\Omega, \Omega'; \Theta, \Theta', \delta \multimap\!\!\otimes\, \delta', \delta'' \multimap\!\!\otimes\, \delta''', \Delta, \Delta', \mathbb{S}, \mathbb{S}' \vdash S \otimes S'}\,(\multimap\!\!\otimes\text{-Merge})$$

For ($\multimap\!\!\otimes$-Spend) and for ($\multimap\!\!\otimes$-Merge) applied to the right derivation, the proof is almost identical.

Take $r' =$ ($\multimap$-left'), and let $r =$ ($\multimap\!\!\otimes$-Spend) be applied to the left derivation

$$\dfrac{\dfrac{\delta \subseteq \delta' \quad \Omega; \Theta, \Delta, \delta', \mathbb{S} \vdash S}{\Omega; \Theta, \delta \multimap\!\!\otimes\, \delta', \Delta, \mathbb{S} \vdash S}\,(\multimap\!\!\otimes\text{-Spend}) \qquad \Omega'; \Theta', \Delta', \mathbb{S}', S' \vdash S''}{\Omega, \Omega'; \Theta, \Theta', \delta \multimap\!\!\otimes\, \delta', \Delta, \Delta', S \multimap S', \mathbb{S}, \mathbb{S}' \vdash S''}\,(\multimap\text{-left'})$$

We can swap the rules as follows.

$$\dfrac{\delta \subseteq \delta' \qquad \dfrac{\Omega;\Theta,\Delta,\delta',\mathbb{S} \vdash S \qquad \Omega';\Theta',\Delta',\mathbb{S}',S' \vdash S''}{\Omega,\Omega';\Theta,\Theta',\Delta,\Delta',\delta',S \multimap S',\mathbb{S},\mathbb{S}' \vdash S''}\,(\multimap\text{-left'})}{\Omega,\Omega';\Theta,\Theta',\delta \multimap \delta',\Delta,\Delta',S \multimap S',\mathbb{S},\mathbb{S}' \vdash S''}\,(\multimap\text{-Spend})$$

For (⊸-Merge) and for (⊸-Spend) applied to the right derivation, the proof is almost identical. □

Recall that Π_A is a derivation applying rules in the set A only.
A CLNL*2 proof is *normalized* if it can be decomposed in

$$\dfrac{\begin{array}{c}\Pi_{\{\text{(L-Ax), (I-right)}\}}\\ \vdots\\ \Pi_{Lr'}\\ \Pi_{Pr}\\ \Pi_{Gr}\\ \Pi_{Cr\cup Sr}\end{array}}{\Omega;\Theta,\Delta,\mathbb{S} \vdash S}$$

Lemma C.8. A sequent $\Omega;\Theta,\Delta,\mathbb{S} \vdash S$ is valid in CLNL*2 if and only if a normalized proof exists for $\Omega;\Theta,\Delta,\mathbb{S} \vdash S$.

Proof. Given a proof Π in CLNL*2 for the sequent, we rewrite it using Lemma C.5 until a fix point is reached, obtaining the following.

$$\dfrac{\begin{array}{c}\Pi_{\{\text{(L-Ax), (I-right)}\}}\\ \vdots\\ \Pi_{Lr'\cup Pr}\\ \Pi_{Gr\cup Cr\cup rS}\end{array}}{\Omega;\Theta,\Delta,\mathbb{S} \vdash S}$$

We rewrite $\Pi_{Gr\cup Cr\cup Sr}$ using Lemma C.6, and $\Pi_{Lr'\cup Pr}$ using Lemma C.7 until a fix point is reached, obtaining a normalized proof. □

We establish now some auxiliary results about reordering rules in CLNL*.

Lemma C.9. If derivation Π with two rules where an application of (⊸-Spend) follows an application of (⊸-Merge) exists in CLNL*, then an equivalent derivation Π' exists where (⊸-Merge) follows (⊸-Spend).

Proof. Let Π be

$$\dfrac{\delta_0 \subseteq \delta_0' \qquad \dfrac{\Omega;\Theta,\delta \otimes \delta'' \multimap \delta' \otimes \delta''',\Delta,\delta_0',\mathbb{S} \vdash S}{\Omega;\Theta,\delta \multimap \delta',\delta'' \multimap \delta''',\Delta,\delta_0',\mathbb{S} \vdash S}\,(\multimap\text{-Merge})}{\Omega;\Theta,\delta_0 \multimap \delta_0',\delta \multimap \delta',\delta'' \multimap \delta''',\Delta,\mathbb{S} \vdash S}\,(\multimap\text{-Spend})$$

The derivation Π' is then as follows.

$$\dfrac{\dfrac{\delta_0 \subseteq \delta_0' \qquad \Omega; \Theta, \delta \otimes \delta'' \multimap \delta' \otimes \delta''', \Delta, \delta_0', \mathbb{S} \vdash S}{\Omega; \Theta, \delta_0 \multimap \delta_0', \delta \otimes \delta'' \multimap \delta' \otimes \delta''', \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Spend})}{\Omega; \Theta, \delta_0 \multimap \delta_0', \delta \multimap \delta', \delta'' \multimap \delta''', \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Merge})$$

□

Lemma C.10. If

$$\dfrac{\delta \subseteq \delta' \qquad \dfrac{\delta'' \subseteq \delta''' \quad \Omega; \Theta, \Delta, \delta', \delta'''\mathbb{S} \vdash S}{\Omega; \Theta, \delta'' \multimap \delta''', \Delta, \delta'\mathbb{S} \vdash S}\,(\multimap\text{-Spend})}{\Omega; \Theta, \delta \multimap \delta', \delta'' \multimap \delta''', \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Spend})$$

is a CLNL* derivation, so is also the following.

$$\dfrac{\dfrac{\delta \otimes \delta'' \subseteq \delta' \otimes \delta''' \quad \Omega; \Theta, \Delta, \delta', \delta'''\mathbb{S} \vdash S}{\Omega; \Theta, \delta \otimes \delta'' \multimap \delta' \otimes \delta''', \Delta\mathbb{S} \vdash S}\,(\multimap\text{-Spend})}{\Omega; \Theta, \delta \multimap \delta', \delta'' \multimap \delta''', \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Merge})$$

Proof. $\delta \subseteq \delta'$ and $\delta'' \subseteq \delta'''$ trivially implies $\delta \otimes \delta'' \subseteq \delta' \otimes \delta'''$. □

We prove now the main result.

Theorem C.1 (normal form)**.** For any $\Omega, \mathbb{S}, S$, the initial sequent $\Omega; \mathbb{S} \vdash S$ is valid in CLNL* if and only if a normal proof Π exists for $\Omega; \mathbb{S} \vdash S$.

Proof. Of course if a normalized proof exists the sequent is valid. Assume $\Omega; \mathbb{S} \vdash S$ is proved by Π_{CLNL*}. First, we rewrite every occurrence of (L-Ax) where $\Omega \neq \emptyset$ as follows

$$\dfrac{\dfrac{}{A \vdash A}\,(\text{L-Ax})}{\Omega; A \vdash A}\,(\text{L-Weak})$$

obtaining the equivalent proof Π'_{CLNL*}.

Then we rewrite Π'_{CLNL*} as an equivalent proof Π_{CLNL*2} in CLNL*2 using Corollary C.1.

By Lemma C.8, a normalized proof Π'_{CLNL*2} exists as follows.

$$\begin{array}{c} \Pi_{\{\text{(L-Ax), (I-right)}\}} \\ \vdots \\ \begin{array}{c} \Pi_{Lr'} \\ \Pi_{Pr} \\ \Pi_{Gr} \\ \Pi_{Cr \cup Sr} \end{array} \\ \vdots \\ \Omega; \mathbb{S} \vdash S \end{array}$$

Since no (L-Weak) rule appears above $\Pi_{Cr \cup Sr}$, and $\Omega = \emptyset$ in the leaves by construction, in the derivation $\Pi_{Lr'}$, the non-linear part of the sequent Ω is $\emptyset$, thus Π'_{CLNL*2} is a CLNL* proof as well, by Lemma C.4.

If Π_P is an empty derivation, then Π'_{CLNL*2} is in first normal form, otherwise Ω_G, Θ, Δ, Δ' exist such that Π'_{CLNL*2} is

$$\dfrac{\Pi_{Lr \cup \{\text{(L-Ax), (I-right)}\}}}{\begin{array}{c}\Delta', \mathbb{S} \vdash S \\ \vdots\ \Pi_{Pr} \\ \Theta, \Delta, \mathbb{S} \vdash S \\ \vdots\ \Pi_{Gr} \\ \Omega_G; \mathbb{S} \vdash S \\ \vdots\ \Pi_{Cr \cup Sr} \\ \Omega; \mathbb{S} \vdash S\end{array}}$$

Then, we rewrite Π_{Pr} using Lemma C.9 until a fix point is reached, obtaining the following.

$$\dfrac{\Pi_{Lr \cup \{\text{(L-Ax), (I-right)}\}}}{\begin{array}{c}\Delta', \mathbb{S} \vdash S \\ \vdots\ \Pi_{(\multimap\text{-Spend})} \\ \vdots\ \Pi_{(\multimap\text{-Merge})} \\ \Theta, \Delta, \mathbb{S} \vdash S \\ \vdots\ \Pi_{Gr} \\ \Omega_G; \mathbb{S} \vdash S \\ \vdots\ \Pi_{Cr \cup Sr} \\ \Omega; \mathbb{S} \vdash S\end{array}}$$

Finally, we rewrite $\Pi_{(\multimap\text{-Spend})}$ using Lemma C.10 until a fix point is reached, obtaining a proof in second normal form. □

C.2.2 Deciding CLNL*

We prove now the decidability of CLNL*. As usual, we start be presenting intermediate results.

Lemma C.11. If a sequent $\Omega; \Theta, \Delta, \mathbb{S} \vdash S$ is derivable from a sequent $\Omega'; \Theta', \Delta', \mathbb{S}' \vdash S'$ only using rules in $C \cup S$, then $\Omega \Vdash \omega$ holds for all $\omega \in \Omega'$.

Proof. By trivial rule induction. □

Lemma C.12. A proof in the first normal form exists for $\Omega; \mathbb{S} \vdash S$ if and only a proof in the first normal form exists for $\Omega_\star; \mathbb{S} \vdash S$ where $\Omega_\star$ contains a single occurence of any $G\delta$ and $G\theta$ such that $\Omega \Vdash G\delta$ and $\Omega \Vdash G\theta$.

Proof. Assume a proof Π exists for $\Omega_\star; \mathbb{S} \vdash S$, and let $\Omega_\star$ be

$$\{G(\theta_i) \mid i \in [1,n]\} \cup \{G(\delta_i) \mid i \in [1,m]\}$$

Then the following proves $\Omega, \mathbb{S} \vdash S$.

$$\dfrac{\Omega \vdash G(\theta_1) \qquad \dfrac{\dfrac{\Omega \vdash G(\delta_1) \quad \dfrac{\dfrac{\Pi}{\Omega_\star; \mathbb{S} \vdash S}}{\vdots\ \Pi_{\{(\text{CL-Cut})\}}} \quad \Omega^{m-1}; G(\theta_1), \ldots, G(\theta_n), G(\delta_1); \mathbb{S} \vdash S}{\Omega^m, G(\theta_1), \ldots, G(\theta_n); \mathbb{S} \vdash S}\,(\text{CL-Cut})}{\vdots\ \Pi_{\{(\text{CL-Cut})\}}} \quad \Omega^{n+m-1} G(\theta_1); \mathbb{S} \vdash S}{\dfrac{\Omega^{n+m}; \mathbb{S} \vdash S}{\Omega; \mathbb{S} \vdash S}\,(\text{L-Cont})}\,(\text{CL-Cut})$$

Assume a proof Π exists for $\Omega; \mathbb{S} \vdash S$ in first normal form as follows

$$\dfrac{\dfrac{\Pi'}{\Omega_G; \mathbb{S} \vdash S}}{\vdots\ \Pi_{Cr \cup Sr}}$$
$$\Omega; \mathbb{S} \vdash S$$

where Π' itself is in first normal form.

By Lemma C.11, all the elements in Ω_G occurs also in $\Omega_\star$ with a single occurrence. We write $\Omega_G = \Omega_\star \cup \Omega_{cont} \setminus \Omega_{weak}$ where Ω_{cont} contains the extra occurences in Ω_G with respect to $\Omega_\star$, and Ω_{weak} contains the elements of $\Omega_\star$ that are not in Ω_G.

Then, the following proof exists for $\Omega_\star; \mathbb{S} \vdash S$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi'}{\Omega_G; \mathbb{S} \vdash S}}{\Omega_\star, \Omega_{cont}; \mathbb{S} \vdash S}\,\text{L-Weak}}{\Omega_\star; \mathbb{S} \vdash S}\,\text{L-Cont}$$

□

We establish some useful results for reducing proofs from second to first normal form.

Lemma C.13. A derivation that only uses rules in $Gr \cup Sr$ exists from $\Theta, \Delta, \mathbb{S} \vdash S$ to $\Omega_G, \mathbb{S} \vdash S$, with

$$\Omega_G = \{G(\theta_i) \mid i \in [1,n]\} \cup \{G(\delta_j) \mid j \in [1,m]\}$$

if and only if $x_1, \dots x_n$ and $z_1, \dots z_m$ nonnegative integers exist such that

$$\Theta = \{\theta_i^{x_i} \mid i \in [1, n]\}$$
$$\Delta = \{\delta_j^{z_j} \mid j \in [1, m]\}$$

Proof. Assume a derivation exists. By trivial rule induction over rules in $Gr \cup Sr$, the linear propositions δ and θ appearing in $\Theta, \Delta, \mathbb{S} \vdash S$ are the same that appears in $\Omega_G, \mathbb{S} \vdash S$ preceded by G, possibly with a different number of occurences. Let x_i and z_j be such occurences. The thesis trivially follows.

Assume Θ and Δ are defined as in the formula above. Let Ω'_G and Ω''_G be

$$\Omega'_G = \{G(\theta_i) \mid \theta_i^{x_i} \in \Theta \wedge x_i \neq 0\} \cup \{G(\delta_j) \mid \delta_j^{z_i} \in \Delta \wedge z_i \neq 0\}$$
$$\Omega''_G = \{G(\theta_i)^{x_i} \mid \theta_i^{x_i} \in \Theta \wedge x_i \neq 0\} \cup \{G(\delta_j)^{z_j} \mid \delta_j^{z_j} \in \Delta \wedge z_j \neq 0\}$$

A derivation exists from $\Theta, \Delta, \mathbb{S} \vdash S$ to $\Omega_G, \mathbb{S} \vdash S$ as follows.

$$\dfrac{\dfrac{\begin{array}{c}\Theta, \Delta, \mathbb{S} \vdash S \\ \vdots\ \Pi_{Gr}\end{array}}{\Omega''_G, \mathbb{S} \vdash S}\,\text{(L-Cont)}}{\Omega'_G, \mathbb{S} \vdash S}\,\text{(L-Weak)}$$

□

Lemma C.14. A derivation that only uses ($\multimap$-Merge) exists from $\theta, \Delta\mathbb{S} \vdash S$ to $\Theta, \Delta\mathbb{S} \vdash S$, with

$$\Theta = \{\delta_i \multimap \delta'_i \mid i \in [0, n]\}$$

if and only if

$$\theta = \bigotimes_{i=1}^{n} \delta_i \multimap \bigotimes_{i=1}^{n} \delta'_i$$

Proof. Trivially derives from the fact that ($\multimap$-Merge) preserves both the multisets of instances of δ that appears on the left side of $\multimap$ and the multiset of the instances of δ that appears on the right side of $\multimap$. □

Lemma C.15. For any $\Omega_G, \Delta_\star, \mathbb{S}, S$, a derivation exists from $\Delta_\star, \mathbb{S} \vdash S$ to $\Omega_G; \mathbb{S} \vdash S$, in one of the following forms

$$\begin{array}{ll} & \dfrac{\Delta_\star, \mathbb{S} \vdash S}{\theta, \Delta, \mathbb{S} \vdash S}\,(\multimap\text{-Spend}) \\ & \quad\vdots\ \Pi_{(\multimap\text{-Merge})} \\ \Delta_\star, \mathbb{S} \vdash S & \Theta, \Delta, \mathbb{S} \vdash S \\ \quad\vdots\ \Pi_{Gr \cup Sr} & \quad\vdots\ \Pi_{Gr \cup Sr} \\ \Omega_G; \mathbb{S} \vdash S & \Omega_G; \mathbb{S} \vdash S \end{array}$$

if and only if $x_1, \dots x_n$ and $z_i, \dots z_m$ nonnegative integers exist such that formulas 4.10 and 4.11 hold.

Proof. Let Ω_G be

$$\Omega_G = \{G(\theta_i) \mid i \in [1,n]\} \cup \{G(\delta_j) \mid j \in [1,m]\}$$

Consider a derivation of the first form. By Lemma C.13, such a derivation exists if and only if $x_1, \dots x_n$ and $z_1, \dots z_m$ nonnegative integers exist such that

$$\begin{aligned} \emptyset = \Theta &= \{\theta_i^{x_i} \mid i \in [1,n]\} \qquad \text{i.e. } x_i = 0 \text{ for all } i \in [1,n] \\ \Delta_\star &= \{\delta_j^{z_j} \mid j \in [1,m]\} \end{aligned}$$

Consider now a derivation of the second form. By Lemma C.13 the derivation $\Pi_{G \cup S}$ exists if and only if $x_1, \dots x_n$ and $z_1, \dots z_m$ nonnegative integers exist such that

$$\begin{aligned} \Theta &= \{\theta_i^{x_i} \mid i \in [1,n]\} \\ \Delta &= \{\delta_j^{z_j} \mid j \in [1,m]\} \end{aligned}$$

Then, by Lemma C.14, the derivation $\Pi_{(\multimap\text{-Merge})}$ exists if and only if

$$\theta = \bigotimes_{j=1}^{m} \delta_j^{z_j} \multimap \bigotimes_{j=1}^{m} (\delta'_j)^{z_j}$$

and by definition of ($\multimap$-Spend) the derivation exists if and only if

$$\Delta_\star = \Delta \cup \{\bigotimes_{j=1}^{m} (\delta'_j)^{z_j}\} \tag{C.1}$$

and

$$\bigotimes_{j=1}^{m} \delta_j^{z_j} \subseteq \bigotimes_{j=1}^{m} (\delta'_j)^{z_j} \tag{C.2}$$

Note that these conditions reduces to the ones of first form when $x_i = 0$ for any $i \in [1,n]$. Thus, we can conclude that a derivation exists if and only if conditions C.1 and C.2 are met.

We conclude by showing that these conditions are equivalent to 4.10 and 4.10 are met respectively.

Recall that $\mathcal{L}_{\Omega_G} = \{\ell_1, \dots \ell_p\}$ is the set of linear implications between atomic propositions appearing as terms in Ω_G. By definition of δ, we can rewrite conditions C.1 and C.2 respectively as follows.

$$\begin{aligned} \Delta_\star = \{(\bigotimes_{k=1}^{p} \ell_k^{A_{k,i}})^{x_i} \mid i \in [1,n]\} \cup \{\bigotimes_{j=1}^{m} (\bigotimes_{k=1}^{p} \ell_k^{C_{k,j}})^{z_j}\} \\ \bigotimes_{j=1}^{m} (\bigotimes_{k=1}^{p} \ell_k^{B_{k,j}})^{z_j} \subseteq \bigotimes_{j=1}^{m} (\bigotimes_{k=1}^{p} \ell_k^{C_{k,j}})^{z_j} \end{aligned}$$

Where for each i, and k, $A_{k,i}$ is the number of occurrences of ℓ_k in δ_i; for each j, and k, $B_{k,j}$ is the number of occurrences of ℓ_k in δ_j, and $C_{k,j}$ is the number of occurrences of ℓ_k in δ'_j. By definition, A_{Ω_G}, contains $A_{k,i}$ in row k, column i; and B_{Ω_G}, and C_{Ω_G} contains $B_{k,j}$ and $C_{k,j}$ in row k, column j respectively. The equivalence between condition 4.10 and C.1 trivially holds.

Take any $z_1, \dots z_m$. Condition C.2 holds if and only if, for any ℓ_k the number of occurrences in the left part of C.2 is grater than the number of occurrences in the right part, i.e.,

$$\bigotimes_{j=1}^{m} (\ell_k^{B_{k,j}})^{z_j} \subseteq \bigotimes_{j=1}^{m} (\ell_k^{C_{k,j}})^{z_j} \qquad \text{for every } \ell_k$$

By definition, this holds if and only if the k-th rows B_k of B_{Ω_G} and C_k of C_{Ω_G} are such that

$$\begin{bmatrix} C_{k,1} & \dots & C_{k,1} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} \geq \begin{bmatrix} B_{k,1} & \dots & B_{k,1} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix}$$

which, in turn, is true for every ℓ_k if and only if condition 4.11 holds. □