SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2014; **00**:1–38
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Formal firewall conformance testing: an application of test and proof techniques

Achim D. Brucker[1,*,†], Lukas Brügger[2] and Burkhart Wolff[3]

[1] *SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany*
[2] *Information Security, ETH Zurich, 8092 Zurich, Switzerland*
[3] *Université Paris-Sud, Laboratoire de Recherche en Informatique (LRI, Bât 650), 91893 Orsay Cedex, France*

SUMMARY

Firewalls are an important means to secure critical ICT infrastructures. As configurable off-the-shelf products, the effectiveness of a firewall crucially depends on both the correctness of the implementation itself as well as the correct configuration. While testing the implementation can be done once by the manufacturer, the configuration needs to be tested for each application individually. This is particularly challenging as the configuration, implementing a firewall policy, is inherently complex, hard to understand, administrated by different stakeholders and thus difficult to validate. This paper presents a formal model of both stateless and stateful firewalls (packet filters), including NAT, to which a specification-based conformance test case generation approach is applied. Furthermore, a verified optimisation technique for this approach is presented: starting from a formal model for stateless firewalls, a collection of semantics-preserving policy transformation rules and an algorithm that optimizes the specification with respect of the number of test cases required for path coverage of the model are derived. We extend an existing approach that integrates verification and testing, that is, tests and proofs to support conformance testing of network policies. The presented approach is supported by a test framework that allows to test actual firewalls using the test cases generated on the basis of the formal model. Finally, a report on several larger case studies is presented. Copyright © 2014 John Wiley & Sons, Ltd.

Received 6 August 2012; Revised 24 June 2014; Accepted 16 July 2014

KEY WORDS: model-based testing; conformance testing; security testing; firewall; specification-based testing; testing cloud infrastructure; transformation for testability; HOL-TESTGEN; test and proof; security configuration testing

1. INTRODUCTION

Because of its connected life, the modern world is increasingly depending on secure implementations and configurations of network infrastructures. As building blocks of the latter, firewalls are playing a central role in ensuring the overall *security* of networked applications.

Firewalls, routers applying NAT and similar networking systems suffer from the same quality problems as other complex software. Jennifer Rexford mentioned in her keynote at POPL 2012 that high-end firewalls consist of more than 20 million lines of code comprising components written in Ada as well as LISP. However, the testing techniques discussed here are of wider interest to all network infrastructure operators that need to ensure the security and reliability of their infrastructures across system changes such as system upgrades or hardware replacements. This is because firewalls and routers are active network elements that can filter and rewrite network traffic based on configurable rules. The *configuration* by appropriate rule sets implements a security policy or links networks together.

*Correspondence to: Achim D. Brucker, SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany.
†E-mail: achim.brucker@sap.com

Thus, it is, firstly, important to test both the implementation of a firewall and, secondly, the correct configuration for each use. While the former can be done once and for all by the firewall vendor, the latter needs to be done by the customer for each and every installation. In practice, an extensive compliance test of firewall configurations is often neglected, even if the security risk of wrong configurations is high:

> NSA found that inappropriate or incorrect security configurations (most often caused by configuration errors at the local base level) were responsible for 80 percent of Air Force vulnerabilities. [16, p. 55]

A particular variant of configuration testing is the test of *collections* of firewalls and routers for their combined behaviour. This is an important scenario for at least two different reasons: first, the security of cloud offerings (for any variant of cloud computing, for example, infrastructure-as-a-service, platform-as-a-service, software-as-a-service) relies on securely separated network segments and thus on correctly configured routers and firewalls. Consequently, the approach discussed in this paper is applicable to any cloud data centre and can support any cloud company in offering secure services to their customers. Second, firewall policies change over time, sometimes modified by different administrators, who may belong to *different* stake-holders enforcing their own security interests. Depending on the concrete testing scenario, a firewall policy can play the role of a program ('Does a firewall interpret its configuration file correctly?') or the role of a policy specification ('Is protocol x possible from a to b?'). For both implementation integration tests as well as configuration tests, the technique presented in this paper can be used to generate runtime testers used in 'hardware-in-the-loop' scenarios during installation and maintenance.

Several approaches for the generation of test cases are well known: while *unit-test*-oriented test generation methods essentially use preconditions and post-conditions of system operation specifications, *sequence-test*-oriented approaches essentially use temporal specifications or automata based specifications of system behaviour. Usually, firewalls combine both aspects: whereas firewall policies are static, the underlying *network protocols* may depend on protocol states which some policies are aware of. This combination of complexity and criticality makes firewalls a challenging and rewarding target for *security testing* in the sense of the ISO 29199 specification [29, Section 4.36], that is, *conformance testing* of security properties. While there is an overlap of the detectable vulnerabilities, this form of conformance testing complements penetration testing approaches that simulate an attacker trying to circumvent security infrastructures.

In this paper, a formal model of firewall policies as packet filtering functions or transformations over them in HOL is presented. Thus, the presented approach can cover data-oriented as well as temporal security policies. On the basis of this formal model, HOL-TESTGEN [7, 11] automatically generates abstract test cases. From the latter, concrete test driver can be derived that can be used for both testing firewall implementations as well as network configurations.

While applying HOL-TESTGEN 'out-of-the-box' to these kind of model works for smaller policies, it does not scale well to policies of practically relevant size, that is, common rule sets with several hundred rules in medium-sized company networks. The situation is particularly worsened if the underlying state is evolving over time (as in stateful firewalls). To overcome this technical difficulty, a policy transformation strategy that transforms a given policy into a functionally equivalent one while improving the effectiveness of test generation by orders of magnitude is developed. The theorem-prover-based approach allows both to formally prove the correctness of this policy transformation, as well as integrating it into the symbolic computations of HOL-TESTGEN.

Summing up, this paper has the following main contributions:

1. a formal model of stateless and stateful firewall policies, instantiating the generic Unified Policy Framework (UPF) [12],
2. a specification-based test generation approach that can be used for testing the correctness of firewall implementations as well as testing that a concrete configuration of a network complies to a given policy specification,
3. a policy transformation approach that is formally proven correct; it is shown to increase the effectiveness of the approach by several orders of magnitude,

4. a combination of unit and sequence test strategies, providing an efficient test technique for stateful firewalls, and
5. a domain-specific test tool built on top of HOL-TESTGEN that increases the degree of automation as well as the effectiveness of HOL-TESTGEN. This test tool provides an infrastructure for test execution and tracing on real firewalls and networks.

This paper extends the previous work of the authors [6, 8, 10, 13, 14] in several aspects: first, partial solutions previously modelled directly in HOL are integrated in the UPF, which allows to highlight common principles. Second, the firewall policy model is extended significantly to include different network models (address representations) as well as pervasive support of NAT. Third, an investigation on the role of the optimisation on the equivalence classes underlying the test generation is added. And, finally, an empirical evaluation covering various aspects of the optimized test generation procedure is provided.

The rest of the paper is structured as follows: after an introduction of the preliminaries, that is, Isabelle/HOL and the HOL-TESTGEN approach (Section 2), we discuss the formal model of stateless firewall policies in Section 3. In Section 5 we extend this model to stateful firewall policies. Section 4 presents the transformation of stateless firewall policies and discusses its effect on the underlying equivalence classes. Before the discussion of empirical results in Section 7, we describe our domain-specific test framework in Section 6. Finally, we discuss related work and draw conclusions in Section 8.

The presented domain-specific test tool as well as all proofs and examples presented in this paper are part of the HOL-TESTGEN distribution, which can be downloaded from the website: http://www.brucker.ch/projects/hol-testgen/.

2. FORMAL AND TECHNICAL BACKGROUND

2.1. *Isabelle and higher-order logic*

Isabelle [34] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic HOL, which is the basis of HOL-TESTGEN and which is introduced in this section. Moreover, Isabelle is also a generic system framework (roughly comparable with Eclipse) which offers editing, modelling, code generation, document generation and (as mentioned) theorem proving facilities, to the extent that some users use it just as programming environment for SML or to write papers over checked mathematical content to generate LaTeX output. Many users know only the theorem proving language Isar for structured proofs and are more or less unaware that this is a particular configuration of the system, that can be easily extended. This is what the HOL-TESTGEN system underlying this work technically is: an extended Isabelle/HOL configuration providing specialized commands for a document-centric model-based test generation method.

Higher-order logic (HOL) [2, 17] is a classical logic based on a simple type system. It provides the usual logical connectives such as $\_ \wedge \_$, $\_ \to \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\ x$ and $\exists x.\ P\ x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centred around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, because, for example, induction schemes can be expressed inside the logic. Being based on a polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language such as SML or Haskell, and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is a logical embedding of HOL into Isabelle. The simple-type system underlying HOL has been extended by Hindley/Milner style polymorphism with type classes similar to Haskell. Isabelle/HOL offers support for a particular methodology to extend given theories in a logically safe way: a theory extension is *conservative* if the extended theory is consistent provided that the original theory was consistent. Conservative extensions can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *well-founded recursive definitions*.

For example, typed sets are built in the Isabelle libraries conservatively on top of the kernel of HOL as functions to bool; consequently, the constant definitions for set comprehension and membership are as follows (to increase readability, the presentation is slightly simplified):

$$\begin{array}{llll} \text{type\_synonym} & \alpha\,\text{set} & = \alpha \Rightarrow \text{bool} & \\ \text{definition} & \text{Collect} & :: (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha\,\text{set} & \text{— set comprehension} \\ \text{where} & \text{Collect}\ S & = S & \\ \text{definition} & \text{member} & :: \alpha \Rightarrow \alpha\,\text{set} \Rightarrow \text{bool} & \text{— membership test} \\ \text{where} & \text{member}\ s\,S & = S\ s & \end{array} \tag{1}$$

Isabelle's powerful syntax engine is instructed to accept the notation $\{x \mid P\}$ for Collect $\lambda\, x.\ P$ and the notation $s \in S$ for member $s\,S$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; these types of axioms are logically safe because they work like an abbreviation. The syntactic side conditions of the axioms are mechanically checked, of course. It is straightforward to express the usual operations on sets such as $\_ \cup \_, \_ \cap \_ :: \alpha\,\text{set} \Rightarrow \alpha\,\text{set} \Rightarrow \alpha\,\text{set}$ as conservative extensions, too, while the rules of typed set theory are derived by proofs from these definitions.

Similarly, a logical compiler is invoked for the following statements introducing the types option and list:

$$\begin{array}{lll} \text{datatype} & \alpha\,\text{option} & = \text{None} \mid \text{Some}\,\alpha \\ \text{datatype} & \alpha\,\text{list} & = \text{Nil} \mid \text{Cons}\ \alpha\ (\alpha\ \text{list}) \end{array} \tag{2}$$

Here, $[]$ and $a\#l$ are alternative syntax for Nil and Cons $a\,l$; moreover, $[a, b, c]$ is defined as alternative syntax for $a\#b\#c\#[]$. Similarly, the option type shown previously is given a different notation: α option is written as $\alpha_{\bot}$, None as $\bot$, and Some X as $\llcorner X \lrcorner$. Internally, recursive datatype definitions are represented by type- and constant definitions. Besides the *constructors* None, Some, Nil and Cons, the aforementioned statement defines implicitly the match-operation case x of $\bot \Rightarrow F \mid \llcorner a \lrcorner \Rightarrow G\,a$, respectively, case x of $[] \Rightarrow F \mid (a\#r) \Rightarrow G\,a\,r$. From the internal definitions (not shown here), many properties are automatically derived like distinctness $[] \neq a\#t$, injectivity of the constructors or induction schemes. Finally, there is a parser for primitive and well-founded recursive function definition syntax. For example, the sort operation can be defined by

$$\begin{array}{lll} \text{fun} & \text{ins} & :: [\alpha :: \text{linorder}, \alpha\ \text{list}] \Rightarrow \alpha\ \text{list} \\ \text{where} & \text{ins}\ x\ [\,] & = [x] \\ & \text{ins}\ x\ (y\#ys) & = \text{if}\ x < y\ \text{then}\ x\#y\#ys\ \text{else}\ y\#(\text{ins}\ x\ ys) \end{array} \tag{3}$$

$$\begin{array}{lll} \text{fun} & \text{sort} & :: (\alpha :: \text{linorder})\ \text{list} \Rightarrow \alpha\ \text{list} \\ \text{where} & \text{sort}\ [\,] & = [\,] \\ & \text{sort}(x\#xs) & = \text{ins}\ x\ (\text{sort}\ xs) \end{array} \tag{4}$$

which is again compiled internally to constant and type definitions. Here, $\alpha :: \text{linorder}$ requires that the type α is a member of the *type class* linorder. Thus, the operation sort works on arbitrary lists of type $(\alpha :: \text{linorder})$ list on which a linear ordering is defined. The internal (non-recursive) constant definition for the operations ins and sort is quite involved and requires a termination proof with respect to a well-founded ordering constructed by a heuristic. Nevertheless, the logical compiler will finally derive all the equations in the aforementioned statements from this definition and makes them available for automated simplification.

On these conservative definition principles, the Isabelle/HOL library provides a large collection of theories such as sets, lists, Cartesian products $\alpha \times \beta$ and disjoint type sums $\alpha + \beta$, multisets, orderings and various arithmetic theories, which only contain rules derived from conservative definitions.

The theory of partial functions is of particular importance for this paper. As prerequisite, we introduce the function $\ulcorner\_\urcorner : \alpha_{\bot} \Rightarrow \alpha$ as the inverse of $\llcorner\_\lrcorner$ (unspecified for $\bot$). Partial functions $\alpha \rightharpoonup \beta$ are then defined as functions $\alpha \Rightarrow \beta_{\bot}$ supporting the usual concepts of domain $\text{dom}\ f \equiv \{x \mid f\ x \neq \bot\}$) and range $\text{ran}\ f \equiv \{x \mid \exists y.\ f\ y = \llcorner x \lrcorner\}$. Partial functions can be viewed as 'maps' or dictionaries; the *empty* map is defined by $\oslash \equiv \lambda\, x.\ \bot$, and the update operation, written $p(x \mapsto t)$, by $\lambda\, y.$ if $y = x$ then $\llcorner t \lrcorner$ else $p\ y$. Finally, the override operation on maps, written $p_1 \oplus p_2$, is defined by $\lambda\, x.$ case $p_1\ x$ of $\bot \Rightarrow p_2\ x \mid \llcorner X \lrcorner \Rightarrow \llcorner X \lrcorner$. This definition incorporates a first-fit-strategy, while the standard Isabelle/HOL operator _ ++_ implements a last-fit strategy.

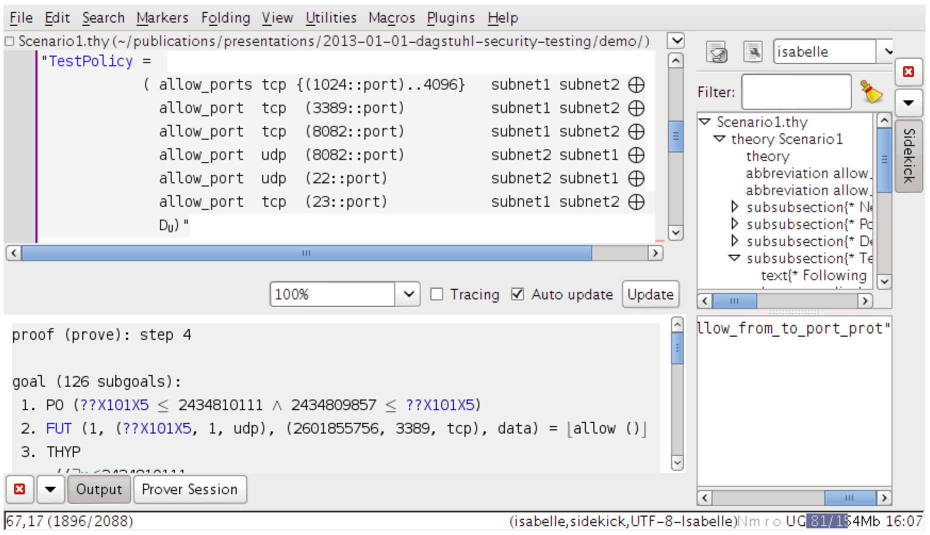
Finally, Isabelle/HOL manages a set of *executable types and operators*, that is, types and operators for which a compilation to SML, OCaml, Scala, or Haskell is possible. Setups for arithmetic types such as int have been done; moreover, any datatype and any recursive function is included in this executable set (providing that they only consist of executable operators). Similarly, Isabelle manages a large set of (higher-order) rewrite rules in which recursive function definitions are included. The Isabelle simplifier provides also a highly potent procedure for computing normal forms of specifications and deciding many fragments of theories underlying the constraints to be processed when constructing test theorems.

2.2. The HOL-TestGen workflow and system architecture

HOL-TESTGEN (Figure 1) is an interactive, that is, semi-automated, test tool for specification-based tests built upon Isabelle/HOL. While Isabelle/HOL is usually seen as 'proof assistant', it is used in this context as a document centric modelling environment for the domain specific background theory of a test (the *test theory*), for stating and logically transforming test goals (the *test specifications*), as well as for the test generation method implemented by Isabelle's tactic procedures. In a nutshell, the test generation method consists of the following:

1. a *test case generation* phase, which is essentially an equivalence partitioning procedure of the input/output relation based on a CNF-like normal form computation,
2. a *test data selection* phase, which essentially uses a combination of constraint solvers using random test generation and the integrated SMT-solver Z3 [18],
3. a *test execution* phase, which reuses the Isabelle/HOL code generators to convert the instantiated test cases to test driver code that is run against a firewall or a network.

A detailed account on the symbolic computation performed by the test case generation and test selection procedures is contained in [7]; its description in detail is out of the scope of this paper. For the remainder of this paper, a high-level view suffices: the test case generation is an *equivalence partitioning* method combined with a variable splitting technique that can be seen as an *(abstract) syntax testing* in the sense of the ISO 29199 specification [29, Sec. 5.2.1 and 5.2.4].


File Edit Search Markers Folding View Utilities Macros Plugins Help
Scenario1.thy (~/publications/presentations/2013-01-01-dagstuhl-security-testing/demo/)
isabelle
Filter:
Sidekick
100%
Tracing
Auto update
Update
Output
Prover Session

```
"TestPolicy =
          ( allow_ports tcp {(1024::port)..4096}   subnet1 subnet2 ⊕
            allow_port  tcp  (3389::port)          subnet1 subnet2 ⊕
            allow_port  tcp  (8082::port)          subnet1 subnet2 ⊕
            allow_port  udp  (8082::port)          subnet2 subnet1 ⊕
            allow_port  udp  (22::port)            subnet2 subnet1 ⊕
            allow_port  tcp  (23::port)            subnet1 subnet2 ⊕
            D_U) "
```

```
Scenario1.thy
  theory Scenario1
    theory
    abbreviation allow_
    abbreviation allow_
    subsubsection{* N
    subsubsection{* P
    subsubsection{* D
    subsubsection{* Te
      text{* Following
```

```
llow_from_to_port_prot"
```

```
proof (prove): step 4

goal (126 subgoals):
 1. PO (??X101X5 ≤ 2434810111 ∧ 2434809857 ≤ ??X101X5)
 2. FUT (1, (??X101X5, 1, udp), (2601855756, 3389, tcp), data) = ⌊allow ()⌋
 3. THYP
```

```
67,17 (1896/2088)        (isabelle,sidekick,UTF-8-Isabelle)Nm r o UG 81/154Mb 16:07
```

Figure 1. A HOL-TESTGEN session showing the jEdit-based Isabelle Interface. The upper-left sub-window allows one to interactively step through a test theory comprising test specifications, while the lower-left sub-window shows the corresponding system state of the spot marked in blue in the upper window.

The equivalence partitioning separates the input/output relation of a programme under test (PUT), usually specified by pre-conditions and post-conditions, into classes for which the tester has reasons to believe that PUT will treat them the same. Technically, this is achieved by converting the test specification into a normal form called TNF^E, which resembles a conjunctive normal form CNF. TNF is an acronym for *Testing Normal Form*, and E is the underling theory of the literals occurring its clause. In our concrete application, E consists of logical rules, arithmetic simplification rules, rules concerning policy simplification and symbolic evaluation (all proven correct with respect to the given theories). Equivalence classes correspond to clauses that are synonym to test cases.

The abstract syntax testing method splits equivalence classes in an interleaved way with the normalization process. Technically, it performs case splitting, which replaces variables in the clauses by terms (so to speak, abstract syntax skeletons, which may again contain variables for holes to be filled in later in the process). For example, a variable of type list is replaced by $[]$, $[a]$, $[a, b]$, and $[a, b, c]@l$ for given test-depth k=3.

Test cases, that is, clauses in normal form, are converted to test data by solving the constraints on input variables of the PUT modulo the underlying theory E. Test cases that are unsatisfiable with respect to E represent empty equivalence classes and are eliminated. Section 4.8 contains a discussion of HOL-TESTGEN's coverage criteria and their relation to policy coverage.

2.3. *The unified policy framework*

The UPF [14] is a generic framework for policy modelling with the primary goal of being used for test case generation. The interested reader is referred to [12] for an application of UPF to large scale access control policies in the health care domain; a comprehensive treatment is also contained in the reference manual coming with the distribution on the HOL-TESTGEN website (http://www.brucker.ch/projects/hol-testgen/). UPF is based on the following four principles:

1. policies are represented as *functions* (rather than relations),
2. policy combination avoids conflicts by construction,
3. the decision type is three-valued (allow, deny, undefined),
4. the output type does not only contain the decision but also a 'slot' for arbitrary result data.

Formally, the concept of a policy is specified as a partial function from some input to a decision value and additional some output. *Partial* functions are used because elementary policies are described by partial system behaviour, which are glued together by operators such as function override and functional composition.

$$\text{type\_synonym} \quad \alpha \mapsto \beta = \alpha \rightharpoonup \beta\, \text{decision} \tag{5}$$

where the enumeration type decision is

$$\text{datatype} \qquad \alpha\ \text{decision} = \text{allow}\ \ \alpha \mid \text{deny}\ \ \alpha \tag{6}$$

As policies are partial functions or 'maps', the notions of a *domain* $\text{dom}\, p :: \alpha \rightharpoonup \beta \Rightarrow \alpha\, \text{set}$ and a *range* $\text{ran}\, p :: [\alpha \rightharpoonup \beta] \Rightarrow \beta\, \text{set}$ can be inherited from the Isabelle library.

Inspired by the Z notation [38], there is the concept of *domain restriction* $\_ \lhd \_$ and *range restriction* $\_ \rhd \_$, defined as:

$$\begin{array}{lll} \text{definition} & \_ \lhd \_ & :: \alpha\,\text{set} \Rightarrow \alpha \mapsto \beta \Rightarrow \alpha \mapsto \beta \\ \text{where} & S \lhd p & = \lambda\, x.\ \ \text{if}\, x \in S\, \text{then}\, p\, x\, \text{else} \perp \\ \text{definition} & \_ \rhd \_ & :: \alpha \mapsto \beta \Rightarrow \beta\ \text{decision set} \Rightarrow \alpha \mapsto \beta \\ \text{where} & p \rhd S & = \lambda\, x.\ \ \text{if}\big(\text{the}\ (p\ x)\big) \in S\, \text{then}\, p\, x\, \text{else} \perp \end{array} \tag{7}$$

The operator 'the' strips off the Some, if it exists. Otherwise the range restriction is underspecified.

There are many operators that change the result of applying the policy to a particular element. The essential one is the *update*:

$$p(x \mapsto t) = \lambda\, y.\ \ \text{if}\, y = x\, \text{then} \lfloor t \rfloor\, \text{else}\, p\ y \tag{8}$$

Next, there are three categories of elementary policies in UPF, relating to the three possible decision values:

- The empty policy; undefined for all elements: $\emptyset = \lambda\, x.\ \bot$
- A policy allowing everything, written as $A_f\ f$, or A_U if the additional output is unit (defined as $\lambda\, x.\ \lfloor \mathrm{allow}() \rfloor$).
- A policy denying everything, written as $D_f\ f$, or D_U if the additional output is unit.

The most often used approach to define individual rules is to define a rule as a refinement of one of the elementary policies, by using a domain restriction. As an example,

$$\big\{(\mathrm{Alice}, \mathrm{obj1}, \mathrm{read})\big\} \lhd A_U \tag{9}$$

Finally, rules can be combined to policies in three different ways:

- Override operators: used for policies of the same type, written as $\_ \oplus_i \_$.
- Parallel combination operators: used for the parallel composition of policies of potentially different type, written as $\_ \otimes_i \_$.
- Sequential combination operators: used for the sequential composition of policies of potentially different type, written as $\_ \circ_i \_$.

All three combinators exist in four variants, depending on how the decisions of the constituent policies are to be combined. For example, the $\_ \otimes_2 \_$ operator is the parallel combination operator where the decision of the second policy is used.

Several interesting algebraic properties are proved for UPF operators. For example, distributivity

$$(P_1 \oplus P_2) \otimes P_3 = (P_1 \otimes P_3) \oplus (P_2 \otimes P_3) \tag{10}$$

Other UPF concepts are introduced in this paper on-the-fly when needed.

3. STATELESS FIREWALLS

This section introduces a formal model of stateless firewall policies as an instantiation of the UPF.

3.1. An introduction to stateless firewalls

A *message* that should be sent from network A to network B is split into several *packets*. A packet contains parts of the content of the message and routing information. The routing information of a packet mainly contains its source and its destination address.

A *stateless firewall*, also called a stateless packet filter, filters (i. e. rejects or allows) traffic from one network to another on the basis of certain criteria as for example on its source and destination address. The *policy* is the specification (or configuration) of the firewall and describes which packets should be denied and which should be allowed.

Table I shows a firewall policy as it can be found in security textbooks for a common firewall setup separating three networks: the external Internet, the internal network that has to be protected (intranet) and an intermediate network, the DMZ. The DMZ is usually used for servers that should be accessible both from the outside (Internet) and from the internal network (intranet) and thus are governed by a more relaxed policy than the intranet. Such a table-based presentation of a policy (Table I) uses a first-fit matching strategy, that is, the first match overrides later ones. For example, a TCP packet from the Internet to the intranet is rejected as it only matches the last line of the table, whereas a TCP packet on port 25 from the intranet to the DMZ is allowed. Lines in such a table are also called *rules*; together, they describe the *policy* of a firewall. In general, a single rule can, conceptually, be understood as a (simple) policy. As the formal model of firewall policies based on UPF provides powerful operators for combining policies, a complex policy can be constructed by combining simpler policies. Thus, the formal model does not need to distinguish policies and rules. Consequently, in this paper, the terms policy and rules are used synonymously.

The rest of this section formalizes the required concepts, that is, networks, addresses, ports and protocols. This formalization provides the basis for formally specifying firewall policies, which can be used for specification-based conformance tests.

Table I. A simple firewall policy separating three networks.

Source	Destination	Protocol	Port	Action
Internet	DMZ	TCP	25	Allow
Internet	DMZ	TCP	80	Allow
Intranet	Internet	UDP	8080	Allow
Intranet	DMZ	TCP	25	Allow
DMZ	Intranet	TCP	25	Allow
Internet	Intranet	UDP	8081	Allow
Any	Any	Any	Any	Deny

Such a policy table uses a first-fit pattern matching strategy, that is, the first match overrides later ones. For example, a TCP packet from the Internet to the intranet is rejected (it only matches the last line of the table), whereas a UDP packet on port 8081 is allowed.

3.2. *A model of networks*

A stateless firewall takes as input a single network packet and statically decides what should be done with it, independently of what has happened before. Thus, all the information common firewalls use to make that decision need to be modelled. To accommodate for the large number of different kinds and models of firewalls, the model is kept generic, also reflecting the need for different kinds of test scenarios.

3.2.1. Network addresses. A network address is used to specify origin and destination of a packet. Depending on the test scenario one has in mind, different kinds of address representations might be desired. While at times, it might be enough to specify the host, at other times, specific host addresses including port numbers are required. For this reason, arbitrary address representation is allowed by declaring the unconstrained type class α adr. The use of a type class allows one to formalize most aspects of the network communication without relying on a concrete representation of network addresses.

Typically, an address consists of a specification of the network and, optionally, a port number and a protocol indication, which can be combined to tuples. Using Isabelle's type class mechanism, there are constants provided for these parts, that further theories can make use of. Address representations that want to use these theories need to provide definitions for those constants.

In the following, some example address representations are provided. First, three possible network representations are presented, which can arbitrarily be combined with a port number and protocol formalization:

- *Datatype:* Many policies do not differentiate the different hosts within a network. Thus, instead of modelling single addresses, it is often sufficient to only specify the network a host is being part of. This can, for example, be done by modelling all the relevant networks as an element of a fixed datatype, as in

$$\text{datatype} \qquad \text{networks} = \text{internet} \mid \text{intranet} \mid \text{dmz} \tag{11}$$

  The advantage of this representation is its simplicity; the state explosion during test case generation can be minimized. An additional post-processing step after test data generation must be used to randomly choose a specific host within a network.
  The expressiblity of the models however, is limited: several hosts within a network cannot be distinguish and the (sub-)networks must not be overlapping. Thus, this model can only be used in rather simple scenarios.
- *IPv4:* Another address format tries to stay as close to reality as possible: IPv4 addresses are directly modelled as a four-tuple of Integers.

$$\text{type\_synonym} \qquad \text{ip} \quad = \text{int} \times \text{int} \times \text{int} \times \text{int} \tag{12}$$

  The expressiveness of the model is significantly extended when using this address representation: each host is specified directly, and thus, overlapping networks can be specified.

Moreover, network masks can be specified directly. This expressiveness comes with a price, however, as also the complexity is larger. Instead of one variable for each host, there are four. This makes test case and test data generation much slower.

- *Integer:* There is a good trade-off between the last two models: modelling an address as a single Integer:

$$\text{type\_synonym} \quad \text{ip}' = \text{int} \tag{13}$$

  This model is as expressive as the IPv4 model: a four-tuple of Integers can always be transformed into a single Integer (after all, an IPv4 address is a 32-Bit Integer). Compared with the IPv4 model, the state space is reduced significantly, which allows for a deeper exploration of the test space.

These network representations can be combined with the concepts of source ports, destination ports and protocols, that is, depending on the purpose of the model, an address can be modelled as an element of a network or it can be refined with either a pair of source and destination ports, a protocol, or both. These concepts can be modelled as follows:

- *Ports:* A common requirement for firewall policy models is port numbers. They can be modelled as Integers.

$$\text{type\_synonym} \quad \text{port} = \text{int} \tag{14}$$

- *Protocols:* If it is required to model a policy that discriminates packets according to the transport layer protocol, those can also be modelled as part of an address. Usually, the number of such protocols is quite small; thus it is best modelled as a datatype:

$$\text{datatype} \quad \text{protocol} = \text{tcp} \mid \text{udp} \tag{15}$$

  At first sight, it might be surprising to model the protocol as part of an address. However, this choice allows to keep the generic model as clean and simple as possible.

For the sake of simplicity, only one address representation is used throughout the rest of this paper: an address is a tuple consisting of two Integers and a protocol indication. The first Integer is denoting a host (encoding its IP address), the second one the port number:

$$\begin{array}{lll} \text{type\_synonym} & \text{address} & = \text{int} \\ \text{datatype} & \text{protocol} & = \text{tcp} \mid \text{udp} \\ \text{type\_synonym} & \text{port} & = \text{int} \\ \text{type\_synonym} & \text{adr}_{\text{IP}} & = \text{address} \times \text{protocol} \times \text{port} \end{array} \tag{16}$$

This representation has turned out to be the most effective one throughout the experiments. Of course, the formalizations and tests work for other representations equally well.

For the sake of simplicity, the same format for the source and destination address of a packet is required. This means that for the protocol field, which needs to be the same for both addresses, only one should be considered and the other one ignored.

3.2.2. Networks. A network (or sub-network) is essentially a set of addresses. To allow some further modelling possibilities, they are defined as sets of sets of addresses—making it simpler to define some networks forming together one bigger network. Thus, a network consists of a set of networks, and a network of a set of addresses:

$$\text{type\_synonym} \quad \alpha\ \text{net} = (\alpha :: \text{adr})\ \text{set set} \tag{17}$$

For checking whether a given address is part of a network, the following operator is defined:

$$\begin{array}{lll} \text{definition} & \_ \sqsubset \_ & :: (\alpha :: \text{adr}) \Rightarrow \alpha\ \text{net} \Rightarrow \text{bool} \\ \quad \text{where} & a \sqsubset S & = \exists s \in S.\ (a \in s) \end{array} \tag{18}$$

3.2.3. Packets. Next to its origin and destination, little more information needs to be encoded into a packet: an identifier (as an Integer) and a generic content. Both are mostly of use later for stateful firewalls and can be set to some default dummy value when restricting testing to stateless firewalls. In the end, a packet is

$$\text{type\_synonym}\ \ (\alpha, \beta)\,\text{packet} = \text{id}\ \times\ (\alpha :: \text{adr})\ \times\ (\alpha :: \text{adr})\ \times\ \beta\,\text{content} \tag{19}$$

Further, there exist projectors, for example, getId, for accessing the components of a packet.

3.3. A model of stateless firewall policies

A firewall policy takes as input a packet and returns a decision value (either allow or deny) and a possibly transformed packet (in the case of NAT).

On the basis of the core language provided by the UPF, domain-specific combinators for defining rules for firewalls and for NAT done by routers are introduced. While some of the rules are for all kinds of packets, others are only applicable for more specific address representations, for example, only those containing port numbers.

3.3.1. Stateless packet filtering rules. A stateless packet filtering rule takes as input a network packet and either allows or denies this packet. There is no address translation or anything similar performed. Thus, the type of such a rule is

$$(\alpha, \beta)\ \text{packet} \mapsto \text{unit} \tag{20}$$

where the type unit consists of only one element written '$()$'. It is used for policies having no special output apart from the decision.

The default operator that denies all traffic can be defined directly (the dual D_U is defined analogously):

$$\begin{array}{ll} \text{definition} & D_U :: (\alpha, \beta)\ \text{packet} \mapsto \text{unit} \\ \text{where} & D_U = \lambda\ x.\ \llcorner\text{deny}\ ()\lrcorner \end{array} \tag{21}$$

Starting from these elementary policies, more complex combinators can be defined. An example of a typical combinator for such a policy is the following one that can be used to model a rule that allows all the traffic between two networks. It is defined by a domain restriction of the allow-everything combinator to those packets having their origin (src_net) and destination ($dest\_net$) in the given networks:

$$\begin{array}{ll} \text{definition} & \text{allow\_from\_to} :: \alpha :: \text{adr net} \Rightarrow \alpha :: \text{adr net} \Rightarrow (\alpha, \beta)\,\text{packet} \mapsto \text{unit} \\ \text{where} & \text{allow\_from\_to}\ \ src\_net\ dest\_net \\ & \quad = \{(pa \mid \text{getSrc}\ \ pa \sqsubset src\_net) \wedge (\text{getDest}\ \ pa \sqsubset dest\_net)\} \lhd \text{allow\_all} \end{array} \tag{22}$$

where getSrc (getDest) allow to access the source (destination) address of a packet. This combinator can then be used to formalize a rule such as

$$\text{allow\_from\_to}\ \ \text{intranet}\ \ \text{internet} \tag{23}$$

A stateless packet filtering policy usually consists of an often very large number of such rules, which are processed in a first-fit manner and a default rule being used when no other rule matches. The operation *override* $\_ \oplus \_ :: \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta$ (cf. Section 2.1) allows elementary policies (*rules*) to be combined to more complex ones in a first-fit manner, for example,

$$p_1 \oplus \cdots \oplus p_n \oplus D_U \tag{24}$$

where the last rule serves as 'catch-all' (for a given function f producing the default return value, if any) that, in this case, denies everything.

The configuration table of Table I is ready to be translated one-to-one into the combinator language of the model just presented:

$$\begin{aligned}\textsf{definition}\quad \text{Policy} \;&= \text{allow\_port\_from\_to internet dmz tcp } 25\\ &\oplus \text{allow\_port\_from\_to internet dmz tcp } 80\\ &\oplus \text{allow\_port\_from\_to intranet internet udp } 8080\\ &\oplus \text{allow\_port\_from\_to intranet dmz tcp } 25\\ &\oplus \text{allow\_port\_from\_to dmz inranet tcp } 993\\ &\oplus \text{allow\_port\_from\_to internet intranet udp } 8081\\ &\oplus D_U\end{aligned} \tag{25}$$

where allow_port_from_to allows traffic from one network to another network only for a specific protocol and destination port. This combinator is defined similarly to allow_from_to (Fact 22).

3.3.2. Network address translation rules. Common firewalls or routers often perform some kind of NAT, where either the source or the destination address of a packet is transformed when passing the firewall. There is a wide range of possible transformations, for example,

- all IP addresses in a given range are transformed to a static one,
- all IP addresses in a given range are transformed (pretty randomly) to a pool of addresses, or
- all IP addresses in a given range are transformed, as well as their ports. The latter is called PAT.

The policy type for these kinds of rules is

$$\textsf{type\_synonym}\quad \text{NAT\_Policy} = (\alpha :: \text{adr}, \beta)\,\text{packet} \mapsto (\alpha, \beta)\,\text{packet set} \tag{26}$$

These rules allow all packets in their domain, but additionally return a set of packets in their decision, which denote the valid packet transformations. A received packet is legitimate if it matches any element of this set. A typical NAT rule, performing NAT to pool transformation on the source address, is

$$\begin{aligned}&\textsf{definition}\quad \text{nat2pool\_IntPort} :: \text{address set} \Rightarrow \text{address set}\\ &\qquad\qquad \Rightarrow (\text{adr}_{\text{IP}}, \beta)\,\text{packet} \mapsto (\text{adr}_{\text{IP}}, \beta)\,\text{packet set}\\ &\textsf{where}\qquad \text{nat2pool\_IntPort } \mathit{srcs}\ \mathit{transl} =\\ &\qquad\qquad \lambda p.\ \text{if } (\text{getSrcAdr}\,p) \in \mathit{srcs}\\ &\qquad\qquad\quad \text{then } \llcorner\text{allow}\{(i, (s_1, s_2), d, c) \mid\\ &\qquad\qquad\qquad\qquad (i = \text{getId}\,p \wedge s_1 \in \mathit{transl} \wedge s_2 = \text{getSrcPort}\,p\\ &\qquad\qquad\qquad\qquad \wedge\, d = \text{getDest}\,p \wedge c = \text{getContent}\,p)\}\lrcorner\\ &\qquad\qquad\quad \text{else } \bot\end{aligned} \tag{27}$$

3.3.3. Combining policies. A real firewall usually performs several activities in parallel, for example, filtering and network address translations. If such a policy needs to be tested, it is necessary to generate test data for a combination of policies of a different type.

In the 'policy-as-function' view, policies are easy to combine using (higher-order) combinators. There are many generic combinators provided for common situations. As an example, there is the parallel combination of two policies, inspired by parallel composition of automata: each policy makes an (independent) step, and the result is a step relation on the Cartesian product of states. There are several different combinators, depending how the filtering decisions are supposed to be combined. Consider, for example, $\_ \otimes_2 \_$ (*parallel-2*), keeping the filtering decision of the second policy only:

$$\begin{array}{ll}
\textsf{definition} & \_ \otimes_2 \_ \; :: (\alpha \mapsto \beta) \Rightarrow (\gamma \mapsto \delta) \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta) \\
\textsf{where} & p_1 \otimes_2 p_2 = \lambda(x, y).\ (\text{case}\, p_1\ x\, \text{of} \\
& \quad \llcorner \text{allow}\ d_1 \lrcorner \Rightarrow (\text{case}\, p_2\ y\, \text{of} \\
& \qquad \llcorner \text{allow}\ d_2 \lrcorner \Rightarrow \llcorner \text{allow}(d_1, d_2) \lrcorner \\
& \qquad \mid \llcorner \text{deny}\ d_2 \lrcorner \Rightarrow \llcorner \text{deny}(d_1, d_2) \lrcorner \\
& \qquad \mid \bot \Rightarrow \bot) \\
& \quad \mid \llcorner \text{deny}\ d_1 \lrcorner \Rightarrow (\text{case}\, p_2\ y\, \text{of} \\
& \qquad \llcorner \text{allow}\ d_2 \lrcorner \Rightarrow \llcorner \text{allow}(d_1, d_2) \lrcorner \\
& \qquad \mid \llcorner \text{deny}\ d_2 \lrcorner \Rightarrow \llcorner \text{deny}(d_1, d_2) \lrcorner \\
& \qquad \mid \bot \Rightarrow \bot) \\
& \quad \mid \bot \Rightarrow \bot)
\end{array} \tag{28}$$

It may be necessary to adapt the input type or output type of a policy to a more refined context. This boils down to variants of functional composition for functions that do not have the format of a policy. With the standard function composition $\_ \circ \_$, it is possible to change the input domain of a policy $p :: (\beta \mapsto \gamma)$ to $p \circ f :: (\alpha \mapsto \gamma)$, provided that f is a coercion function of type $\alpha \Rightarrow \beta$. The same effect can be achieved for the range with the $\circ_f$-operator.

These combination combinators can be used to combine a stateless packet filtering policy with a policy governing network address translation. Assume the policies have the following type:

$$\text{Filter} :: (\text{adr}_{\text{IP}}, \beta)\ \text{packet} \mapsto \text{unit} \tag{29}$$

and a network address translation policy of type:

$$\text{NAT} :: (\text{adr}_{\text{IP}}, \beta)\ \text{packet} \mapsto ((\text{adr}_{\text{IP}}, \beta)\ \text{packet})\ \text{set} \tag{30}$$

Their combination is

$$\begin{array}{lll}
\textsf{definition} & \text{Policy} & :: (\text{adr}_{\text{IP}}, \beta)\ \text{packet} \mapsto ((\text{adr}_{\text{IP}}, \beta)\ \text{packet})\ \text{set} \\
\textsf{where} & \text{Policy} & = (\lambda(x, y).\ x) \circ_f (\text{NAT} \otimes_2 \text{Filter}) \circ (\lambda\, x.\ (x, x))
\end{array} \tag{31}$$

The first coercion function throws away the return value of the second policy (which is just unit), the second one assures that instead of a pair of packets for the input, only one is kept.

3.4. *Testing stateless firewalls*

The *test specification* for the stateless firewall case is now within reach: for every packet x, the *firewall under test (FUT)* has the same behaviour as specified in the just formalized policy:

$$\textsf{testspec}\ \text{test:} \qquad FUT\ x = \text{Policy}\ \ x \tag{32}$$

Usually, this test specification is extended by predicates ensuring that only valid test data (e. g. no negative numbers in the IP addresses) will be generated, and that the content fields will be set to some default value. Another predicate can ensure that only packets which cross network boundaries are considered.

Before applying the test case generation algorithm, (recall Section 2.2), the test specification needs to be transformed. The granularity of the test space partitioning is determined by how much of the definition are unfolded before the algorithm is applied; for example, the networks could be kept closed, or completely unfolded. Much more test data would be produced in the latter case.

The algorithm will return a set of abstract test cases. Test data generation instantiates each of them to concrete values using constraint and random solving mechanisms. Next are two examples of test data for a filtering policy, where the first is a packet meant to be rejected by the firewall, while the second one should be passed:

1. $FUT\ (12, (2, 1, \text{tcp}), (1002, 80, \text{tcp}), \text{content}) = \llcorner \text{deny}\ () \lrcorner$
2. $FUT\ (8, (1002, 1, \text{udp}), (2, 8080, \text{udp}), \text{content}) = \llcorner \text{allow}\ () \lrcorner$

Such test data can be exported and used as input for a tool that executes the tests on a real network setup (the test driver). This is shown in Section 6. Overall, testing stateless firewalls is quite similar to classical unit testing of stateless software.

4. NORMALIZING POLICIES

A common characteristic of 'real' policies is that they consist of a large number of rules. To overcome the inherent state explosion of the underlying model-exploration technique, a few new optimisation ideas have to be added; as will be seen later, these will also help to overcome the apparent drawback of using a powerful modelling language such as HOL, which is more distant to efficient computation and deduction than, say, a limited form of first-order logic with suitable instrumentation ('triggers') for, say, an SMT solver such as Z3 [18].

A key observation is that semantically equivalent policies can be formulated in many syntactically different ways. Moreover, parts of them can be redundant. Consider the following case, where $\mathrm{allow\_port\_from\_to}$ is the combinator allowing all packets of a specific protocol on a specific destination port between two networks:

$$\begin{array}{l}\textbf{lemma}\ \mathrm{redundant\_allowPort:}\\ \quad \mathrm{deny\_from\_to}\ x\ y \oplus \mathrm{allow\_port\_from\_to}\ x\ y\ pr\ po = \mathrm{deny\_from\_to}\ x\ y\end{array} \tag{33}$$

The expression on the left corresponds roughly to a nesting of if-then-else-expressions where the inner branches of the second branch are never reached: consequently they represent unfeasible parts that would, if not eliminated beforehand, eventually result in path conditions (in the model) that have to be proven unfeasible in the general CNF-normalization procedure. The experiments show that in practice these situations occur often in various different disguises.

Of course, the rule shown previously could be used directly by the Isabelle simplifier (a rewriting-based decision procedure); however, it turns out that a certain strategy in applying these and similar rules is necessary and a non-deterministic simplification procedure is insufficient for this purposes.

In the following, a normalization procedure for firewall policies is presented that consists of nine different policy transformation procedures. First, an abstract syntax for the firewall policy language is presented, and next transformations on policies are reformulated as a particular syntactic transformation process. The advantage of this technique—sometimes called reification in the literature—is that local transformations on the syntax as well as the entire normalization strategy can be defined *inside* HOL, as an ordinary function definition, and semantical correctness of local transformations as well as the entire normalization can be *verified* by formal proof.

4.1. Reification of policies and correct policy transformations

For the transformations, an abstract syntax called *policy syntax* is introduced. For the sake of conciseness, only a few core combinators are presented here.

$$\begin{array}{lrl}\textbf{datatype} & (\alpha,\beta)\,\mathrm{Comb} = & \mathrm{DenyAll}\\ & | & \mathrm{DenyFromTo}\ \alpha\ \alpha\\ & & \vdots\\ & | & \mathrm{AllowPort}\ \alpha\ \alpha\ \beta\\ & | & \mathrm{Conc}\ ((\alpha,\beta)\,\mathrm{Comb})\ ((\alpha,\beta)\,\mathrm{Comb})\end{array} \tag{34}$$

The infix notation $\_ \oplus_S \_$ is used for the constructor $\mathrm{Conc}\ \_\ \_$. This abstract syntax is linked to the semantics described in the previous sections by a semantic interpretation function C:

$$\begin{array}{ll}\multicolumn{2}{l}{\textbf{fun}\ \mathrm{C} :: (\mathrm{adr_{IP}}\ \mathrm{net}, \mathrm{protocol} \times \mathrm{port})\,\mathrm{Comb} \Rightarrow (\mathrm{adr_{IP}}, \mathrm{Content})\,\mathrm{packet} \mapsto \mathrm{unit}}\\ \multicolumn{2}{l}{\textbf{where}}\\ \quad \mathrm{C}\ \mathrm{DenyAll} & = D_U\\ \quad \mathrm{C}\ (\mathrm{DenyFromTo}\ x\ y) & = \mathrm{deny\_from\_to}\ x\ y\\ \quad\quad \vdots & \vdots \quad\quad \vdots\\ \quad \mathrm{C}\ (\mathrm{AllowPort}\ x\ y\ (pr, po)) & = \mathrm{allow\_port\_from\_to}\ x\ y\ pr\ po\\ \quad \mathrm{C}\ (x \oplus_S y) & = \mathrm{C}\ x \oplus \mathrm{C}\ y\end{array} \tag{35}$$

In the following, the concept of a policy transformation as well as the semantic correctness of a policy transformation is introduced:

Definition 4.1 (Policy transformation)
A *policy transformation* T for a policy syntax $P :: (\alpha, \beta)\,\mathrm{Comb}$ is a function taking P as input and returning a list $[P_1, \ldots, P_n] :: (\alpha, \beta)\,\mathrm{Comb\,list}$, which is interpreted as $P_1 \oplus_S \cdots \oplus_S P_n$, and where the domains of all P_i are pairwise disjoint, with the possible exception of P_n.

Definition 4.2 (Semantic correctness of policy transformations)
A policy transformation T is *correct* if and only if for all policy syntax $P :: (\alpha, \beta)\,\mathrm{Comb}$ it holds:

$$\begin{aligned} \mathrm{C}\ P \quad &= \mathrm{foldl}\ (\_ \oplus \_)\ \varnothing\ (\mathrm{map}\ \mathrm{C}\ (T\ P)) \\ &= \varnothing \oplus (\mathrm{C}\ P_1) \oplus \cdots \oplus (\mathrm{C}\ P_n) \quad \text{where } T\ P = [P_1, \ldots, P_n], \end{aligned} \tag{36}$$

using the standard map and fold functions on lists and the empty map $\varnothing$. As abbreviation for transforming a list of policy syntax into a single policy, the following operator is defined

$$\text{definition list2policy } l = \mathrm{foldl}\ (\_ \oplus \_)\ \varnothing\ l \tag{37}$$

The normalization procedure is constructed from correct transformation rules only; it is a particular advantage of the approach that the correctness of the entire procedure can be, and is, formally verified.

4.2. *Technique of normalization*

There are in general three kinds of strategies a (correct) transformation procedure might employ to ensure the transformed policy is better suited for test case generation:

1. Eliminate redundant rules: rules that are shadowed by others are never fired and can therefore be removed from the model as redundant.
2. Combine similar rules to a single one: for example, instead of a sequence of rules for one particular element, a single rule might allow a range containing all these elements.
3. Pre-partition the test space: often, policies consist of several independent parts, which might be processed individually. For example, rules P_1 and P_2 might *partially* overlap, such that cutting the domain into only P_1, only P_2, and mixed is advantageous.

The first point can be justified in all cases semantically, because it is easy to see that it does not even change the test cases or partitions because only vacuous (empty) test partitions are removed. This is in contrast to the second one, where the price has to be payed that test partitions are joined and therefore (because for each non-vacuous partition, one test is selected) the coverage with respect to the original policy is reduced. The last point, finally, is very domain-specific and must be applied with care.

As mentioned earlier, only correct transformations are used. For a number of rules, however, correctness can only be established under some preconditions of the target to be transformed; for this reason, the overall normalization must be organized into a total of nine phases, where each phase establishes stronger invariants than the previous one. This also explains why an organization into just a set of rewriting rules to be put in the Isabelle simplifier is impossible and must be replaced by a process that applies these rules according to a strategy.

Furthermore, the transformation procedure is presented in a way such that it can be integrated into the HOL-TESTGEN methodology. To achieve a pre-partitioning of the test space, the test specification must be rewritten into a set of test specifications, where each element considers only one part of the original specification. Technically, this can be achieved by rewriting the original test specification into a conjunction, which is later transformed into a sequence of subgoals, each processed individually by the test case generation algorithm. This technique is only correct if it can be assured that each test specification is creating test cases for the relevant domain only. From this fact stems the requirement that the domains of the sub-policies need to be pairwise distinct.

Recall that in general the format of a test specification is as follows:

$$P\ x \Longrightarrow FUT\ x = \mathrm{Policy}\ x \tag{38}$$

First, the following function is required, which transforms the conclusion of such a test specification into a conjunction of individual specifications if provided a list of policies.

$$\begin{array}{lll} \text{fun} & \text{FUTList} & :: (\alpha \mapsto \beta) \Rightarrow \alpha \Rightarrow (\alpha \mapsto \beta)\ \text{list} \Rightarrow \text{bool}\ \ \text{where} \\ & \text{FUTList}\ \ FUT\ x (p \# ps) & = (x \in \text{dom}\, p \longrightarrow (FUT\ x = p\, x)) \\ & & \quad \wedge (\text{FUTList}\ \ FUT\ x\ ps) \\ & \text{FUTList}\ \ FUT\ x\ [] & = \text{true} \end{array} \tag{39}$$

The theorem

$$\begin{array}{l} \text{theorem distrFUTL:} \\ \quad \text{assumes a1:}\ \ x \in \text{dom}\ P\ \ \text{and a2:}\ \ (\text{list2policy}\ \ PL)\ = P \\ \quad \text{and a3:}\ \ \text{FUTList}\ \ FUT\ x\ PL \\ \text{shows} \qquad FUT\ x = P\ x \end{array} \tag{40}$$

is most crucial for the entire technique: it can be applied in a test specification to execute the transformation of the test specification into a conjunction of test specifications. Because this is stated and proved as a theorem, the transformation is guaranteed to be correct, given that two assumptions (preconditions) hold:

(a1) the domain of the original policy must be universal (the policy must be defined everywhere, that is, not yielding $\perp$) and

(a2) the semantical equivalence of the original policy with the list of sub-policies as stated in Fact 36 must hold.

For the common special case where there is a default rule at the end of the policy, an alternative version of FUTList exists, ensuring that the domain of the last sub-policy is the universe minus the domains of the other sub-policies.

In the following, first some elementary transformation rules are presented, followed by a detailed description of one notable transformation procedures. Next, a complete normalization procedure for stateless firewall policies consisting of a couple of transformation functions applied in sequence is presented. Finally, the effect of the normalization procedure on a small example is presented. The section concludes with observations on coverage.

4.3. *Elementary transformation rules*

A large collection of elementary policy transformation rules can be proved correct. For example, it can be shown that a shadowed rule is redundant, the $\_ \oplus_S \_$ operator is associative, and in many cases, commutativity holds:

$$\text{C}\ (\text{DenyAll} \oplus_S a) = \text{C}\ \ \text{DenyAll} \tag{41}$$

$$\text{C}((a \oplus_S b) \oplus_S c) = \text{C}(a \oplus_S (b \oplus_S c)) \tag{42}$$

$$\text{C}(\text{AllowPort}\ \ x\ y\ a \oplus_S \text{AllowPort}\ \ x\ y\ b) = \text{C}(\text{AllowPort}\ \ x\ y\ b \oplus_S \text{AllowPort}\ \ x\ y\ a) \tag{43}$$

$$\begin{array}{l}\text{C}(\text{DenyFromTo}\ \ x\ y \oplus_S \text{DenyFromTo}\ \ u\ v) \\ \quad = \text{C}(\text{DenyFromTo}\ \ u\ v \oplus_S \text{DenyFromTo}\ \ x\ y)\end{array} \tag{44}$$

$$\text{dom}(\text{C}\ \ a) \cap \text{dom}(\text{C}\ \ b) = \{\} \Longrightarrow \text{C}(a \oplus_S b) = \text{C}(b \oplus_S a) \tag{45}$$

The proofs of these theorems are trivial to establish by the simplifier provided in Isabelle.

4.4. *A transformation procedure*

Here, one particular transformation procedure is presented. The procedure takes a list of single rules and a list defining an ordering on networks as input, and returns an ordered list consisting of the same rules.

The sorting phase is formalized as a function mapping a list of syntactic policy operators (called combinators) to a list of combinators sorted according to the following principles:

- If there is a $\mathrm{DenyAll}$, it should be at the last place.
- Rules dealing with the same set of networks should be grouped together, with the $\mathrm{DenyAll}$'s coming after the $\mathrm{AllowPort}$'s.

These principles motivate a pre-ordering, which is not unique in general, but which can be completed to an ordering (satisfying reflexivity, transitivity and anti-symmetry) under certain side conditions to be discussed later. Formally, a smaller relation is defined on policy syntax:

$$\begin{array}{lll}
\mathbf{fun} & \mathrm{smaller} & :: (\alpha\,\mathrm{set})\,\mathrm{list} \Rightarrow (\alpha,\beta)\,\mathrm{Comb} \Rightarrow (\alpha,\beta)\,\mathrm{Comb} \Rightarrow \mathrm{bool}\ \mathbf{where} \\
 & \mathrm{smaller}\ l\ x\ \mathrm{DenyAll} & = (x \neq D_U) \\
 & \mathrm{smaller}\ l\ \mathrm{DenyAll}\ x & = \mathrm{false} \\
 & \mathrm{smaller}\ l\ x\ y & = (x = y) \vee \\
 & & \quad \mathrm{if}\ \mathrm{bothNet}\ x = \mathrm{bothNet}\ y \\
 & & \quad \mathrm{then}\ \mathrm{case}\, x\ \mathrm{of} \\
 & & \qquad \mathrm{DenyFromTo}\ a\ b \Rightarrow y = \mathrm{DenyFromTo}\ b\ a \\
 & & \qquad \mid \_ \Rightarrow \mathrm{true} \\
 & & \quad \mathrm{else}\ \big(\mathrm{pos}\ (\mathrm{bothNet}\ x)\ l\big) \leq \big(\mathrm{pos}\ (\mathrm{bothNet}\ y)\ l\big)
\end{array} \tag{46}$$

Here, $\mathrm{bothNet}$ returns the set of the source and destination network of a rule, and pos returns the position of an element within a list. The variable l in the aforementioned definition is a list of network sets. The ordering of this list determines which group of rules is treated as smaller than another. For the correctness of the transformation, the concrete ordering of this list is irrelevant, but it is required that all the sets of network pairs of a policy are in this list. This requirement is formalized in a predicate called $\mathrm{all\_in\_list}$. There are circumstances not directly related to testing, where the ordering of this list might become relevant. This would allow one to transform a policy into a semantically equivalent one, where certain classes of rules (for example, those dealing with the most frequently appearing traffic) will be considered earlier by the firewall.

Instead of $\mathrm{smaller}\ \ l\ \ x\ \ y$ it is also possible to write $x \leq_l y$. With this ordering, a version of the sorting function can be supplied—similar to Fact 3—with the ordering as explicit argument $\mathrm{sort}\ \ \leq_l\ \ S$; this represents the core algorithm of the transformation phase. Of course, in practice, a more efficient algorithm than the just insertion sort shown in this presentation is chosen.

All the proofs about the semantical correctness of the sorting transformation, some of which are outlined in the succeeding text, merely require the fact that the sorting algorithm returns a sorted list, such that any sorting algorithm may be used as long as it can be proved that it is returning a list that is sorted according to the aforementioned sorted definition. One of the main prerequisites for such a proof is to show that the ordering relation is transitive, reflexive and anti-symmetric. This is true as long as all the network pairs that appear in the policy are in the list l.

Next, one has to prove that the sorting transformation is semantics-preserving. The following assumptions are made about the input policy:

- $\mathrm{singleComb}$: the policy is given as a list of single rules.
- $\mathrm{allNetsDistinct}$: all the networks are either equal or disjoint.
- $\mathrm{wellformed\_policy1}$: there is exactly one $\mathrm{DenyAll}$ and it is at the last position.
- $\mathrm{wellformed\_policy3}$: the domain of any $\mathrm{AllowPortFromTo}$ rule is disjoint from all the rules appearing on the left of it.

In the next section, it is shown that these conditions always hold at that specific point in the normalization algorithm. To facilitate the proofs, there exists a matching function that returns $\llcorner r \lrcorner$ for the first rule r in a policy p that matches packet x and $\perp$ if there is no such rule. To prove the semantic correctness of sort, that is to prove that the application of the sort transformation does not change the semantics of the policy, an indirect approach is applied. First, it is shown that for all packets x, and for all policies p and s that satisfy the conditions previously listed, and that have the same set of rules (but not necessarily in the same order), the first rule that matches x in p is the same as in s.

$$\begin{array}{ll}
\textsf{theorem}\ \text{C\_eq\_Sets\_mr:} & \\
\quad \textsf{assumes}\ \text{sets\_eq:}\ \ \mathrm{set}\,p = \mathrm{set}\,s & \\
\quad \textsf{and}\ \text{wp1\_p:}\ \ \mathrm{wellformed\_policy1}\ p & \textsf{and}\ \text{wp1\_s:}\ \ \mathrm{wellformed\_policy1}\ s \\
\quad \textsf{and}\ \text{wp3\_p:}\ \ \mathrm{wellformed\_policy3}\ p & \textsf{and}\ \text{wp3\_s:}\ \ \mathrm{wellformed\_policy3}\ s \\
\quad \textsf{and}\ \text{aND:}\ \ \mathrm{allNetsDistinct}\ p & \\
\quad \textsf{and}\ \text{SC:}\ \ \mathrm{singleComb}\ p & \\
\textsf{shows}\ \ \mathrm{matching\_rule}\ x\ p = \mathrm{matching\_rule}\ x\ s &
\end{array} \tag{47}$$

In the succeeding text, a proof sketch of the aforementioned theorem (Fact 47) is presented, which is also proven formally in Isabelle. A case distinction over $\mathrm{matching\_rule}\ x\ p$ results in two cases: this expression can either be $\perp$ or $\llcorner y \lrcorner$:

1. $\perp$. This case is shown by contradiction. As there is a $\mathrm{DenyAll}$, which matches all packets, the matching rule of x cannot be $\perp$.
2. $\llcorner y \lrcorner$. This is shown by case distinction on y, that is, on the matching rule. By definition of the Comb datatype, there are four possibilities:
   - $\mathrm{DenyAll}$: if the matching rule is $\mathrm{DenyAll}$, there is no other rule in p that matches, as it is necessarily the rule considered last. Thus, as the two sets are equal, there is no other rule in s that matches either. As $\mathrm{DenyAll}$ is in s, this has to be the matching rule in s.
   - $\mathrm{DenyFromTo}\ \ a\ b$: if this is the matching rule in p, there cannot be another rule in p that matches: as the networks are disjoint and the only $\mathrm{DenyAll}$ is the last considered rule, the only possibility is an $\mathrm{AllowPort}\ \ a\ \ b\ \ c$, for some c. However, because of $\mathrm{wellformed\_policy3}$, such a rule can only be on the left of the matching rule as their domains are not disjoint. But if it were on the left, the allow-rule would be the matching rule. Thus, by contradiction, there is no such rule. Because such a rule does not exist in p, it does not exist in s either. As $\mathrm{DenyFromTo}\ \ a\ b$ is in s, this has to be the matching rule in s.
   - $\mathrm{AllowPort}\ \ a\ b\ c$: the proof of this subgoal is very similar to the previous one: the only alternative rule that could match is the corresponding $\mathrm{DenyFromTo}\ \ a\ b$, but if this rule appears on the left of the matching rule; this would contradict the wellformedness conditions. Therefore, this has to be the matching rule also in s.
   - $a \oplus_S b$: this case is ruled out by the wellformedness conditions.

The formal proof of this lemma can be found in the appendix. Having proofs that any sorted list of rules conforms to the conditions after sorting if the input list does, the main correctness theorem of the sorting transformation is established:

$$\begin{array}{l}
\textsf{lemma}\ \text{C\_eq\_sorted:} \\
\quad \textsf{assumes}\ \text{ail:}\ \ \mathrm{all\_in\_list}\ \ p\ l \\
\quad \textsf{and}\ \text{wp1:}\ \ \mathrm{wellformed\_policy1}\ \ p \\
\quad \textsf{and}\ \text{wp3:}\ \ \mathrm{wellformed\_policy3}\ \ p \\
\quad \textsf{and}\ \text{aND:}\ \ \mathrm{allNetsDistinct}\ \ p \\
\quad \textsf{and}\ \text{SC:}\ \ \mathrm{singleComb}\ \ p \\
\quad \textsf{shows}\ \ \mathrm{C}\ (\mathrm{list2policy}(\mathrm{sort}\ (\leq_p)\ l)) = \mathrm{C}(\mathrm{list2policy}\ \ p)
\end{array}$$

Thus, there is a sorting transformation for policies that is proven to be semantics-preserving given that some well-specified conditions hold for the input policy.

4.5. *The normalization procedure*

The sorting transformation is one part of a full normalization procedure that is presented next. The procedure is organized into nine phases in total.

The result of the nine phases is a list of policies, in which:

- each element of the list contains a policy that completely specifies the blocking behaviour between two networks, and
- there are no shadowed rules.

This result is desirable because the test case generation for rules between networks A and B is independent of the rules that specify the behaviour for traffic flowing between networks C and D. Thus, the different segments of the policy can be processed individually. The normalization procedure does not aim at minimizing the number of rules. While it does remove unnecessary ones, it also adds new ones, enabling a policy to be split into several independent parts.

The following two restrictions are imposed on the input policies:

- Each policy must contain a $\mathrm{DenyAll}$ rule. If this restriction was to be lifted, the $\mathrm{insertDenies}$ phase would have to be adjusted accordingly.
- For each pair of networks n_1 and n_2, the networks are either disjoint or equal. If this restriction was to be lifted, some additional phases would be required before the start of the normalization procedure presented in the succeeding text. This rule would split single rules into several rules by splitting up the networks such that they are all pairwise disjoint or equal. Such a transformation is clearly semantics-preserving and the condition would hold after these phases.

The full procedure is defined as follows:

$$\begin{aligned}\textbf{definition}\ \mathrm{normalize}\, p = (&\mathrm{removeAllDuplicates} \circ \mathrm{insertDenies}\\ &\circ \mathrm{separate} \circ (\mathrm{sort}\ \ \leq_{\mathrm{Nets\_List}}\ p)\\ &\circ \mathrm{removeShadowRules2} \circ \mathrm{remdups}\\ &\circ \mathrm{removeShadowRules3} \circ \mathrm{insertDeny}\\ &\circ \mathrm{removeShadowRules1} \circ \mathrm{policy2list})\ p\end{aligned} \tag{48}$$

The nine transformation phases of the normalization procedure and their preconditions are next discussed in more detail. The preconditions are necessary to establish the correctness theorem for the overall procedure:

$$\begin{aligned}&\textbf{theorem}\ \mathrm{C\_eq\_normalize}:\\ &\quad \textbf{assumes}\ \mathrm{a1}:\ \ \mathrm{member}\ \ \mathrm{DenyAll}\ \ p\quad \textbf{and}\ \mathrm{a2}:\ \ \mathrm{allNetsDistinct}\ \ p\\ &\textbf{shows}\quad \mathrm{C}(\mathrm{list2policy}(\mathrm{normalize}\ \ p)) = \mathrm{C}\ \ p\end{aligned} \tag{49}$$

The proof of this theorem combines the local correctness results of the single transformations (e. g. the sorting phase result described in the previous section) while discharging their preconditions using invariance properties of the previous transformations. The nine transformations are as follows:

1. **policy2list**: transforms a policy into a list of single rules. The result does not contain policies composed via $\_ \oplus_S \_$, that is, the property $\mathrm{singleComb}$ is established.
2. **removeShadowRules1**: removes all the rules that appear after a $\mathrm{DenyAll}$. The transformation preserves $\mathrm{singleComb}$ and establishes $\mathrm{wellformed\_policy1}$, stating that there is a $\mathrm{DenyAll}$ at the end of the policy.
3. **removeShadowRules3**: removes all rules with an empty domain; they never match any packet. It maintains all previous invariants and does not establish any required new one.
4. **remdups**: removes duplicate rules. Only the first rule appearing in the list remains. As policies are evaluated from left to right, it can easily be shown that this transformation preserves the semantics.
5. **removeShadowRules2**: removes all rules allowing the traffic on a specific port between two networks, where an earlier rule already denies all the traffic between them. This function maintains the earlier invariants, and establishes $\mathrm{wellformed\_policy3}$, the invariant necessary for the sorting function to be correct. It ensures that the domain of an $\mathrm{AllowPortFromTo}$ rule is disjoint from all the rules appearing on the left of it.
6. **sort**: maintains the previous invariants and, besides the sorted invariant, also establishes an important new one: the rules are grouped according to the two networks they consider.

7. **separate**: transforms the list of single combinators into a list of policies. Each of those policies contains all the rules between two networks (in both directions). While $\mathrm{singleComb}$ is invalidated by this transformation, all the others are maintained. Two important additional properties hold after this transformation:
   - $\mathrm{OnlyTwoNets}$: each policy treats at most two different networks.
   - $\mathrm{separated}$: the domains of the policies are pairwise disjoint (except from $\mathrm{DenyAll}$). This follows from the fact that the rules were already grouped.
8. **insertDenies**: is the only phase where additional rules are inserted. At the end of each policy, the two rules denying all traffic between those two networks in both directions are added. As new rules are added here, the proof of semantic equivalence is relatively involved. The main part consists of proving the property that $\mathrm{separated}$ is maintained with this transformation. This phase is only semantics-preserving because of the initial requirement that the policy has a $\mathrm{DenyAll}$. Only after this phase the traffic between two networks is characterized completely by the corresponding policy.
9. **removeAllDuplicates**: removes superfluous duplicate $\mathrm{DenyFromTos}$ introduced by the previous phase.

After the last phase, the result is a list of policies that satisfies the requirements stated in the beginning of this section. Using the correctness of the transformations, it is straightforward to prove semantic equivalence of the full normalization. In the end, the individual elements of the returned list are policies on their own and can be processed individually by HOL-TESTGEN. Thus, the set of test cases for the full policy is decomposed into the set of test cases of the smaller policies.

4.6. *Example*

In the following, the effect of the normalization procedure on the running example policy (Fact 25) is presented. Recall that the original policy contains seven rules restricting the traffic between three networks; After applying the normalization procedure, there are the following policies:

1. $\mathrm{AllowPort\ intranet\ dmz\ (tcp, 25)} \oplus_S \mathrm{AllowPort\ dmz\ intranet\ (tcp, 25)}$
   $\oplus_S \mathrm{DenyFromTo\ dmz\ intranet} \oplus_S \mathrm{DenyFromTo\ intranet\ dmz}$
2. $\mathrm{AllowPort\ intranet\ internet\ (udp, 8080)} \oplus_S \mathrm{AllowPort\ internet\ intranet\ (udp, 8081)}$
   $\oplus_S \mathrm{DenyFromTo\ intranet\ internet} \oplus_S \mathrm{DenyFromTo\ internet\ intranet}$
3. $\mathrm{AllowPort\ internet\ dmz\ (tcp, 80)} \oplus_S \mathrm{AllowPort\ internet\ dmz\ (tcp, 25)}$
   $\oplus_S \mathrm{DenyFromTo\ internet\ dmz} \oplus_S \mathrm{DenyFromTo\ dmz\ internet}$
4. $\mathrm{DenyAll}$

Summing up, the process results in four policies with a total of 13 rules. The number of generated test cases shrinks from 60 to 12, the required time reduces from 400 to about 2 seconds (including normalization).

4.7. *Distributivity of transformations*

Firewall policies are often created by a parallel composition of smaller policies. As this might lead to a large state explosion hindering efficient test case generation, the question arises how transformations can be used together with parallel compositions.

Of course, it is possible to devise a transformation procedure for a parallel combined policy. However, this is in many cases not easy achievable and not compatible with the goal of a modular modelling approach. Much better would be to have transformation procedures for the smaller policy parts and the means to transfer these transformations over parallel compositions.

Transformation procedures can be distributed over parallel compositions such as $\_ \otimes_2 \_$ (Fact 28) thanks to distributivity laws holding over these operators:

$$(P_1 \oplus P_2) \otimes_2 Q \quad = (P_1 \otimes_2 Q) \oplus (P_2 \otimes_2 Q) \tag{50}$$

Assume a policy under test being created as a combination of policies P and Q, and a correct transformation T transforming P into the list $[P_1, P_2, \ldots, P_N]$. Then, for policies that are combined in parallel such as $P \otimes_2 Q$, the following theorem holds:

$$(\mathrm{C}\, P) \otimes_2 Q = \mathrm{list2policy}[\mathrm{C}\, P_1 \otimes_2 Q, \ldots, \mathrm{C}\, P_N \otimes_2 Q]$$

This gives rise to a transformation strategy for policies consisting of many parts executed in parallel: define transformation procedures for the individual policies, which in the end will lead to a transformed combined policy. Of course, the number of final sub-policies might grow very large when transformation procedures are being executed on several constituent policies, as all the sub-policies are paired. For this reason, it is sensible to have transformations which are similar in the sense that they create similar domains for their sub-policies. In that way, most domains of the combined sub-policies are empty, and those elements can be eliminated later.

4.8. *Excursion: test coverage, test size and policy normalization effects*

HOL-TESTGEN is centred around the idea to construct a test for each feasible (i. e. satisfiable) test case; this corresponds to a one-to-one non-empty equivalence class coverage in the sense of the ISO 29199 specification [29, par 5.2.1.3, case 1.a]. The standard states explicitly its unawareness on syntax testing coverage criteria; in the case of HOL-TESTGEN, [7] propose the notion of a one-to-one-coverage to the normal form TNF_d^E, where d is the abstract syntax tree depth up to which variables belonging to a given set of types are split.

The question arises, even if policies are shown to be semantically equivalent, what is the impact on the set of generated tests? Experiments (Section 7) reveal that the tests generated from policies and their optimized counterparts are obviously not identical; it can therefore not be taken for granted that the equivalence partitioning before and after normalization is related.

To shed some light on the nature of the optimizing effect of policy normalization, the effect of the generic equivalence partitioning process in the specific domain of security policies is studied. The simplest UPF constructors for elementary UPF policies are $\varnothing$, $p(x \mapsto t)$, and $p_1 \oplus p_2$. A policy p like $\varnothing(x_1 \mapsto t_1)(x_2 \mapsto t_2)(x_3 \mapsto t_3)$ is converted by unfolding definitions (Section 2.1) to

$$\lambda\, y.\ \text{if } y = x_1 \text{ then } \llcorner t_1 \lrcorner \text{ else if } y = x_2 \text{ then } \llcorner t_2 \lrcorner \text{ else if } y = x_3 \text{ then } \llcorner t_3 \lrcorner \text{ else } \bot \tag{51}$$

that is, a cascade of nested conditional expressions. As part of the background theory E, test case generation uses case split rules for conditionals such as

$$P(\text{if } C \text{ then } D \text{ else } E) \equiv (C \to P(D) \wedge (\neg C \to P(D)) \tag{52}$$

which brings the aforementioned Fact 51 for the test specification $p\ z = r$ into the form:

$$\begin{array}{ll} z = x_1 & \to \llcorner t_1 \lrcorner = r\ \wedge \\ z \neq x_1 \wedge z = x_2 & \to \llcorner t_2 \lrcorner = r\ \wedge \\ z \neq x_1 \wedge z \neq x_2 \wedge z = x_3 & \to \llcorner t_3 \lrcorner = r\ \wedge \\ z \neq x_1 \wedge z \neq x_2 \wedge z \neq x_3 & \to \bot = r \end{array} \tag{53}$$

which is close to a CNF with four conjoints, that is, four test cases. Domain-specific policy combinators such as allow_port_from_to are also built from conditionals, only that the condition is significantly more complex and involves arithmetic and set theoretic reasoning. For the override, the conditionals of the one policy are crafted into the final else branch of the other; this gives rise to the following simple meta-function that provides a tight bound for the number of test cases implied by a policy: $|[\![p(x \mapsto t)]\!]| = |[\![p]\!]| + 1$ and $|[\![p_1 \oplus p_2]\!]| = |[\![p_1]\!]| + |[\![p_2]\!]|$. It is easy to check that the operators of the product family such as $\_ \otimes_2 \_$ and the sequential policy compositions family such as $\_ \circ_p \_$ behave multiplicative on test cases, which explains the exponential effect over the length of test sequences over stateful policies to be discussed in the next section.

From this perspective, it is easy to see that a policy represents (large) nested conditional expressions, and HOL-TESTGEN's CNF-computation constructs all path conditions in the model. The generic normal form TNF_d^E implies for $d = 2$ for a policy $\alpha \mapsto \beta$ path coverage, provided that β is simple and not again a datatype requiring case splitting. Thus, for small d and a fixed E (essentially, HOL + UPF + domain-specific combinator theory), the result is a natural interpretation of the generated tests. The *test hypothesis* ([7] and [22]) underlying the test selection are somewhat 'reasonable' assumptions. This is especially true for the *uniformity hypothesis* (here, pick one instance of the path condition; if the test passes, the system conforms to the specification for all instances of this path). For the *regularity hypothesis,* there is no need because for a stateless firewall, the involved data structures of policies are all non-recursive.

After explaining the construction of the equivalence classes, that is, the test cases, as clauses in the CNF-derivation of policies, the question how the optimization affects their number and relative size can be addressed. The following situations may occur:

1. a policy is shadowed or a duplicate. This corresponds to empty equivalence classes, that is, conjoints that reduce to true because the premise of the implications become false in the underlying theory E. This is the most common case, which does not change the number of feasible test cases *at all.*
2. two policies p_1 and p_2 are 'adjacent' equivalence classes, that is, there is a more general one p_x that subsumes both and the union of the domains of both are exactly $\mathrm{dom}\, p_x$. In this case, the number of test cases decreases by one.
3. two policies p_1 and p_2 overlap, that is, $\mathrm{dom}\, p_1 \cap \mathrm{dom}\, p_1 \neq \{\}$. In this case, policy splitting may be applied that transform $p_1 \oplus p_2$ into $p'_1 \oplus p'_2 \oplus p'_3$ such that the domains of the p'_i are distinct. The number of test cases increases by one.

The key observation is now that the latter situation called *policy split* may only arise a small and bounded number of times during the process.

Generally speaking, the number of policy splits depends on the number of networks and the number of policy rules R. In particular, the number of splits grows, firstly, at most linearly with the number of networks S, that is, specified networks are never split into smaller units. Secondly, every rule of a policy can at most overlap with all other rules. Thus, the number of policy splits grows at most with N^2 of the initial policy. In case there are more overlapping rules than networks S, there must be shadowed or duplicates rules. Thus, in this case the overall number of policy splits is at most equal to the number of networks S. Similarly, if there are more networks as overlapping rules, only the overlapping rules are split.

Thus, the number of equivalence classes grows after normalization in the worst case by at most $\min(S, N^2)$ and decreases in the average case substantially. In most practical scenarios, a policy contains much more rules than networks, thus the number of splits is usually restricted by S.

Summing up, the selected test data may be different, but the equivalence partitioning after optimization is related in the following way: in general, it is a coarser partitioning in the sense that non-empty partitions in original policy are either maintained or fused with adjacent partitions during optimization. However, there is a usually small number of exceptions where an equivalence class is split into smaller classes during this process.

5. STATEFUL FIREWALLS

The stateless scenario just introduced does not solve all security problems of modern network infrastructures. There are several network protocols that require the firewall to observe the internal state of a protocol run, and therefore require a *stateful firewall.* A stateful firewall keeps track of the existing connections and can adapt its filtering behaviour accordingly. As an example, the well-known FTP protocol is based on a dynamic negotiation of a port number, which is then used as channel to communicate the file content between the sender and the receiver. Thus, a stateless firewall can only provide a very limited form of network protection if protocols such as FTP are involved, whereas a stateful firewall observing the inner state of FTP sessions can open the negotiated port dynamically. Similarly, modern VoIP protocols as well as protocols used for streaming multimedia data use dynamically negotiated protocol parameters for optimizing the network load. Testing stateful firewalls, where the filter functions change, requires test sequence generation and the means to model and reason about states and state transitions.

In the following, the method for sequence testing in the HOL-TESTGEN framework is presented. For this purpose, first the foundational setting of test sequences in monads are presented, followed by how to represent stateful policies as abstract specifications, how to model test purposes to slice down the input trace space and finally how to generate test cases for this.

The stateful FTP policy is modelled in a deterministic way: the client explicitly asks for a port number. This behaviour is called *active FTP* in the protocol specification [36]. Because of the lack of space, the presentation is limited to this protocol variant. Ways to model other variants are described in [6], where FUT-controlled non-deterministic behaviour is represented by explicit variables that get replaced on the fly by concrete values during test execution. An alternative would be to admit the Hilbert-choice operator $\mathrm{Some}\, x.\ P\ x$ that chooses an arbitrary x satisfying P if it exists, and an undefined value otherwise. In that model, however, the constraint-resolution/data selection can no longer be done completely during the test case generation (hence, statically), rather it must be finished at test execution time, where observed behaviour leads to new constraints that eventually should eliminate Hilbert-choices in the constraint-list of a test run on the fly. A detailed description of this technique called online testing is out of the scope of this paper.

5.1. Sequence testing in HOL

As HOL is a purely functional specification formalism, it has no built-in concepts for states and state transitions. Using *monads*—a concept made popular for purely functional programming languages by [35]—is one way to overcome this apparent limitation. Abstractly, a monad is a type constructor with a unit and a bind operator enjoying unit and associativity properties. Because of well-known limitations of the Hindley-Milner type system, it is not possible to represent monads *as such* in HOL, only concrete instances. The state-exception monad is one such instance best fitted for this purpose, because it precisely models partial state transition functions of type

$$\text{type\_synonym} \qquad (o,\sigma)\,\mathrm{MON}_{\mathrm{SE}} = \sigma \rightharpoonup (o \times \sigma) \tag{54}$$

Using monads, programmes under test, PUT, can be seen as *i/o stepping functions* of type $\iota \Rightarrow (o,\sigma)\,\mathrm{MON}_{\mathrm{SE}}$, where each stepping function may either fail for a given state σ and input ι, or produce an output o and a successor state.

The usual concepts of *bind* (representing sequential composition with value passing) and *unit* (representing the embedding of a value into a computation) are defined for the case of the state-exception monad as follows:

$$\begin{array}{lll} \text{definition} & \mathrm{bind}_{\mathrm{SE}} & :: (o,\sigma)\,\mathrm{MON}_{\mathrm{SE}} \Rightarrow (o \Rightarrow (o',\sigma)\,\mathrm{MON}_{\mathrm{SE}}) \Rightarrow (o',\sigma)\,\mathrm{MON}_{\mathrm{SE}} \\ \text{where} & \mathrm{bind}_{\mathrm{SE}}\, f g & = \lambda\,\sigma.\ \mathrm{case}\, f\sigma\ \mathrm{of}\ \bot \Rightarrow \bot \\ & & \qquad\qquad\qquad \mid \llcorner(out,\sigma')\lrcorner \Rightarrow g\ out\ \sigma' \end{array} \tag{55}$$

and

$$\begin{array}{lll} \text{definition} & \mathrm{unit}_{\mathrm{SE}} & :: o \Rightarrow (o,\sigma)\,\mathrm{MON}_{\mathrm{SE}} \\ \text{where} & \mathrm{unit}_{\mathrm{SE}}\, e & = \lambda\,\sigma.\ \llcorner(e,\sigma)\lrcorner \end{array} \tag{56}$$

$x \leftarrow f; g$ is written for $\mathrm{bind}_{\mathrm{SE}}\, f (\lambda\, x.\ g)$ and return for $\mathrm{unit}_{\mathrm{SE}}$. On this basis, the concept of a *valid test sequence* (no exception, P yields true for observed output) can be specified:

$$\sigma \models o_1 \leftarrow PUT\ i_1; \ldots; o_n \leftarrow PUT\ i_n; \mathrm{return}(P\, o_1\ \cdots\ o_n) \tag{57}$$

where $\sigma \models m$ is defined as $(m\,\sigma \neq \bot \wedge \mathrm{fst}(\ulcorner m\,\sigma \urcorner)$. For iterations of i/o stepping functions, an mbind operator can be used, which takes a list of inputs $\iota s = [i_1,\ldots,i_n]$, feeds it subsequently into PUT and stops when an error occurs. Using mbind, valid test sequences for a programme under test (PUT) satisfying a post-condition P can be reformulated by

$$\sigma \models os \leftarrow \mathrm{mbind}\ \iota s\ PUT; \mathrm{return}(P\ os) \tag{58}$$

which is the standard way to represent sequence test specifications in HOL-TESTGEN. The construction can be used both for specification purposes (in this case, the behaviour of PUT must be constrained by a specification and $P\ os$ is typically of the form $os = X$ where X is a free variable for instances of valid runs of PUT) as well as code included into the test driver and run in the test execution. In the latter case, PUT must be a free variable marked in the test case generation phase to designate the *programme under test*.

5.2. A stateful firewall model

In general, policies based on a state σ are modelled as follows:

- Model the policy behaviour based on a given state.
- Model state transitions as a partial function to be applied whenever the policy allows an access.
- Model state transitions to be applied whenever the policy denies an access.
- Use the UPF operator $\_ \otimes_{\nabla} \_$ to combine the three parts in parallel.

The operator $\_ \otimes_{\nabla} \_$ is defined as follows:

$$\begin{array}{llcl}
\text{definition} & \_ \otimes_{\nabla} \_ & :: & (\alpha \rightharpoonup \gamma) \times (\alpha \rightharpoonup \gamma) \Rightarrow \delta \mapsto \beta \Rightarrow (\delta \times \alpha) \mapsto (\beta \times \gamma) \\
\text{where} & (A, D) \otimes_{\nabla} p & = & \lambda\, x.\, \text{case}\, p(\text{fst}\ x)\, \text{of} \\
& & & \quad \llcorner\text{allow}\ y\lrcorner \Rightarrow (\text{case}(A(\text{snd}\, x))\ \text{of} \\
& & & \qquad\qquad \bot \Rightarrow \bot \\
& & & \qquad\qquad |\llcorner z\lrcorner \Rightarrow \llcorner\text{allow}(y, z)\lrcorner) \\
& & & \quad |\llcorner\text{deny}\ y\lrcorner \Rightarrow (\text{case}(D(\text{snd}\, x))\ \text{of} \\
& & & \qquad\qquad \bot \Rightarrow \bot \\
& & & \qquad\qquad |\llcorner z\lrcorner \Rightarrow \llcorner\text{deny}(y, z)\lrcorner) \\
& & & \quad |\bot \Rightarrow \bot
\end{array} \tag{59}$$

In the end, a stateful policy, also called a *transition policy*, has the following format:

$$(\iota \times \sigma) \mapsto (o \times \sigma) \tag{60}$$

Such transition policies are isomorphic to state-exception monads introduced before and there are two conversion functions linking them. For example, to transform a transition policy into a state-exception monad, the following operator can be used:

$$\begin{array}{llcl}
\text{definition} & \text{p2M} & :: & (\iota \times \sigma) \mapsto (o \times \sigma) \Rightarrow \iota \Rightarrow (o\,\text{decision}, \sigma)\,\text{MON}_{\text{SE}} \\
\text{where} & \text{p2M}\ p & = & \lambda\ \iota\, \sigma.\, \text{case}\, p(\iota, \sigma)\, \text{of} \\
& & & \llcorner\text{allow}(o, \sigma')\lrcorner \Rightarrow \llcorner(\text{allow}\ o, \sigma')\lrcorner \\
& & & |\llcorner\text{deny}(o, \sigma')\lrcorner\ \Rightarrow \llcorner(\text{deny}\ o, \sigma')\lrcorner \\
& & & |\bot \Rightarrow \bot
\end{array} \tag{61}$$

It is easy to check that p2M is a bijection to state exception monads.

The question arises how to instantiate the state σ in question. It needs to contain the currently active policy (which can change dynamically), and 'other stuff' captured by the type variable α. Thus:

$$\text{type\_synonym} \qquad (\alpha, \beta, \gamma)\,\text{FwState} = \alpha \times (\beta, \gamma)\,\text{packet} \mapsto \text{unit} \tag{62}$$

and the resulting state transition notion modelled as a partial function:

$$\begin{array}{ll}
\text{type\_synonym} & (\alpha, \beta, \gamma)\,\text{FWStateTransition} = \\
& (\beta, \gamma)\,\text{packet} \times (\alpha, \beta, \gamma)\,\text{FWState}) \rightharpoonup (\alpha, \beta, \gamma)\,\text{FWState}
\end{array} \tag{63}$$

The most general way to refine the 'other stuff' α is to instantiate it by the history, that is, the list of so far received packets. Of course, in firewall implementations, complex data structures such as tables keeping the state of protocol runs for each conversation may be possible; however, these should be constructible via abstraction functions from the history. Because the goal is to aim at the most general concept of a stateful policy specification, an instance for α is defined as:

$$\text{type\_synonym} \qquad (\beta, \gamma)\,\text{history} = (\beta, \gamma)\,\text{packet list} \tag{64}$$

It is now possible to introduce a wealth of specification formalisms to describe properties on the history; because HOL is used as background formalism, there is nearly arbitrary freedom here. In the theories, for example, there exists an LTL like language with the operators *atom*: $«\_» :: (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha \text{ list} \Rightarrow \text{bool}$, *next*: $\text{N} :: (\alpha \text{ list} \Rightarrow \text{bool}) \Rightarrow \alpha \text{ list} \Rightarrow \text{bool}$, *always*: $\Box\_ :: (\alpha \text{ list} \Rightarrow \text{bool}) \Rightarrow \alpha \text{ list} \Rightarrow \text{bool}$ and its dual *eventually*, as well as the operator *until*: $\_ \text{ U } \_ :: (\alpha \text{ list} \Rightarrow \text{bool}) \Rightarrow (\alpha \text{ list} \Rightarrow \text{bool}) \Rightarrow \alpha \text{ list} \Rightarrow \text{bool}$. For reasons of space limitations, this theory is not developed here, with the exception of one special operator:

$$\begin{aligned}
\textsf{fun}\quad & \text{before} && :: (\alpha \Rightarrow \text{bool}) \Rightarrow (\alpha \Rightarrow \text{bool}) \Rightarrow \alpha \text{ list} \Rightarrow \text{bool} \ \ \textsf{where} \\
& \text{before } p\, q\, [] && = \text{false} \\
& \text{before } p\, q\, (a \# S) && = q\, a \vee (p\, a \wedge (\text{before } p\, q\, S))
\end{aligned} \tag{65}$$

which can be processed by the HOL-TESTGEN procedure directly and which is formally proven equivalent to $\text{before } p \ q = ((\Box «p») \text{ U } «q»)$.

5.3. *Modelling the file transfer protocol (FTP)*

At this point, the model of the stateful FTP protocol is outlined, as a typical example of stateful network protocols.

During an FTP session, the server opens a data connection to the client using a port that was indicated by the client. Figure 2 shows an abstract trace of an FTP session: the client initialises the session by sending an init message to the server, next the client sends a port_request containing a dynamic port for the data connection, and then the server sends the data to the client using this dynamic port. Eventually, the client will close the connection and the firewall will block the data port. The messages are identified using the content part of a packet. More formally, there is:

$$\textsf{datatype}\, \text{msg} = \text{init} \mid \text{port\_request}\, \textit{port} \mid \text{data} \mid \text{close} \mid \text{other} \tag{66}$$

The id part of a packet is used to distinguish FTP sessions. Next, the state transitions have to be defined. The case for denial state transition is trivial since the state is unchanged:

$$\begin{aligned}
\textsf{fun}\quad & D_{ST} :: ((\text{adr}_{\text{IP}}, \text{msg}) \text{ history}, \text{adr}_{\text{IP}}, \text{msg}) \text{ FWStateTransition} \\
\textsf{where}\quad & D_{ST}(p, s) = \llcorner s \lrcorner
\end{aligned} \tag{67}$$

For the acceptance state transition, three cases must be distinguished.

1. If a legitimate port request message arrives, the policy is extended by a rule that opens the given port, and the packet is appended to the history.
2. Several sender IP's s may get the right to communicate via a port p. However, the policy ensures as invariant that the triple of session id i, sender s and open port p is unique at any time. If all senders have closed their protocol along a port, the policy will deny any further communication along this port.

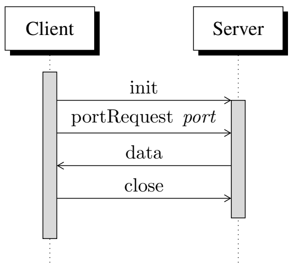


Figure 2. An abstract trace of an FTP session: the client initialises the session by sending an init message to the server; next the client sends a port_request containing a dynamic port for the data connection, and then the server sends the data to the client using this dynamic port. Eventually, the client will close the connection, and the firewall will block the data port.

3. All other messages do not change the policy, but the packets are appended to the history.

Those three cases are modelled as follows:

$$
\begin{array}{l}
\textsf{fun}\ \mathrm{A}_{ST} :: ((\mathrm{adr_{IP}}, \mathrm{msg})\ \mathrm{history}, \mathrm{adr_{IP}}, \mathrm{msg})\ \mathrm{FWStateTransition} \\
\textsf{where} \\
\ \mathrm{A}_{ST}((i, s, d, \mathrm{port\_request}\ pr), (\mathit{log}, \mathit{policy})) = \\
\quad \mathrm{if\ before(Not\ o\ is\_close}\ i)(\mathrm{is\_init}\ i)(\mathit{log}) \wedge \\
\qquad \mathrm{get\_dest\_port}(i, s, d, \mathrm{port\_request}\ pr) = 21 \wedge \mathrm{get\_protocol}\ d = \mathrm{tcp} \\
\quad \mathrm{then}\ \llcorner((i, s, d, \mathrm{port\_request}\ pr)\#\mathit{log}, \\
\qquad\quad \mathrm{allow\_port\_from\_to}(\mathrm{net\_of}\ d)(\mathrm{net\_of}\ s)(pr) \oplus \mathit{policy})\lrcorner \\
\quad \mathrm{else}\ \llcorner((i, s, d, \mathrm{port\_request}\ pr)\#\mathit{log}, \mathit{policy})\lrcorner \\
\mid \mathrm{A}_{ST}((i, s, d, \mathrm{close}), (\mathit{log}, \mathit{policy})) = \\
\quad \mathrm{if}(\exists p.\ \mathrm{same\_open\_ports}\ \ \mathit{log}\ i\ s = [(p, i, s)]) \wedge \mathrm{get\_dest\_port}(i, s, d, \mathrm{close}) = 21 \wedge \\
\qquad \mathrm{get\_protocol}\ d = \mathrm{tcp} \\
\quad \mathrm{then}\llcorner((i, s, d, \mathrm{close})\#\mathit{log}, \\
\qquad \mathrm{deny\_from\_to}\ (\mathrm{net\_of}\ d)\ (\mathrm{net\_of}\ s)\ (\mathrm{fst}(\mathrm{hd}(\mathrm{same\_open\_ports}\ \ \mathit{log}\ i\ s))) \\
\qquad\quad \oplus \mathit{policy})\lrcorner \\
\quad \mathrm{else}\llcorner((i, s, d, \mathrm{close})\#\mathit{log}, \mathit{policy})\lrcorner \\
\mid \mathrm{A}_{ST}(p, (\mathit{log}, \mathit{policy})) = \llcorner(p\#\mathit{log}, \mathit{policy})\lrcorner
\end{array}
\tag{68}
$$

The auxiliary function same_open_ports (not formally defined here) extracts from the log the last port request of s for session i, and filters from the log the triples of s' and i' with this p that constitute open connections. Next, the policy is modelled. The policy that needs to applied in any given state is part of that state. Thus the policy is the application of the policy of the current state. This is formalized as follows:

$$\textsf{definition}\ \mathrm{applyPolicy} :: (\iota \times (\iota \mapsto \alpha)) \mapsto \alpha\ \ \textsf{where}\ \mathrm{applyPolicy} = \lambda(x, z).z\ x \tag{69}$$

Finally, the state transitions are combined with the policy:

$$
\begin{array}{l}
\textsf{definition}\ \mathrm{policy} :: \\
\qquad (\mathrm{adr_{IP}}, \mathrm{msg})\ \mathrm{packet} \times (\mathrm{adr_{IP}}, \mathrm{msg})\ \mathrm{history} \times ((\mathrm{adr_{IP}}, \mathrm{msg})\ \mathrm{packet} \mapsto \mathrm{unit}) \\
\quad \mapsto \mathrm{unit} \qquad\qquad\qquad \times (\mathrm{adr_{IP}}, \mathrm{msg})\ \mathrm{history} \times ((\mathrm{adr_{IP}}, \mathrm{msg})\ \mathrm{packet} \mapsto \mathrm{unit}) \\
\textsf{where} \quad \mathrm{policy} = ((A_{ST}, D_{ST}) \otimes_{\nabla} \mathrm{applyPolicy})(\lambda(x, y).((x, (\mathrm{snd}\ y)), (x, y)))
\end{array}
\tag{70}
$$

where the function $(\lambda(x, y). \ldots)$ serves for a mere technical *repackaging* of the underlying state and input formats involved in the composition. The modelled ftp policy is a *policy transforming policy*, or *second order policy* as can be inferred from the type; with similar techniques, *administrative* policies such as administrative RBAC policies [37] can be modelled and tested in the framework. Furthermore note that the policy specifies a machine that accepts arbitrary interleaved runs of the protocol; for practical purposes, this generality has to be restricted by *test purposes* discussed in the next section. Finally, note that this transition policy can be transformed to a monad by using p2M (as will be done in the test specification).

5.4. *Test purposes for FTP*

From this model, test sequences for FTP packets could be generated directly, producing sequences of more or less random packets. This is usually not what one wants to test against, and thus further means for constraining the test space according to the need of the test engineer are required. There are a couple of standard test purposes when testing for stateful network protocols:

- Tests for one correct protocol run.
- Tests for several correct protocol runs sequentially.
- Tests for several correct protocol runs interleaved.
- Tests for one or several illegal protocol runs.
- Tests for protocol runs interspersed with packets that do not belong to a protocol run.

In the following, only simple correct protocol runs are presented. In the framework, test purposes are constraints (pre-conditions) of test specifications that can be added at wish and that generate (when symbolically evaluated during test case generation) constraint systems that have a similar effect than a product-automata construction, albeit avoiding any finiteness-assumptions.

Because a simple correct run can be in any of four different states, the following datatype is introduced:

$$\text{datatype} \qquad \text{ftp\_states} = \mathrm{S}_0 \mid \mathrm{S}_1 \mid \mathrm{S}_2 \mid \mathrm{S}_3 \tag{71}$$

The following recursive definition checks for legal protocol runs

$$\begin{aligned}
&\text{fun} \quad \text{is\_ftp} :: \text{ftp\_states} \Rightarrow \text{adr}_{\text{IP}} \Rightarrow \text{adr}_{\text{IP}} \Rightarrow \text{id} \Rightarrow \text{port} \Rightarrow \\
&\qquad\quad (\text{adr}_{\text{IP}}, \text{msg})\ \text{history} \Rightarrow \text{bool} \\
&\text{where} \\
&\qquad \text{is\_ftp}\, H\, c\, s\, i\, p\, [\,] = (H = S_3) \\
&\qquad \text{is\_ftp}\, H\, c\, s\, i\, p\, (x\#t) = \\
&\qquad (\lambda(id, sr, de, co).\ (((id = i \wedge \wedge\, \text{get\_dest\_protocol}(id, sr, de, co) = \text{tcp} \wedge (\\
&\qquad\quad (H = S_2 \wedge sr = c \wedge de = s \wedge co = \text{init} \wedge \text{is\_ftp}\, S_3\, c\, s\, i\, p\, t) \vee \\
&\qquad\quad (H = S_1 \wedge sr = c \wedge de = s \wedge co = \text{port\_request}\, p \wedge \text{is\_ftp}\, S_2\, c\, s\, i\, p\, t) \vee \\
&\qquad\quad (H = S_1 \wedge sr = s \wedge de = (\text{fst}\, c, p) \wedge co = \text{data} \wedge \text{is\_ftp}\ S_1\, c\, s\, i\, p\, t) \vee \\
&\qquad\quad (H = S_0 \wedge sr = c \wedge de = s \wedge co = \text{close} \wedge \text{is\_ftp}\, S_1\, c\, s\, i\, p\, t)))))\ x
\end{aligned} \tag{72}$$

Finally, everything is wrapped together and the single_run-predicate is defined for traces, restricting the set of traces to legal runs of the ftp protocol and, furthermore, restricting the used network addresses to traffic between specific networks:

$$\begin{aligned}
\text{definition} \quad \text{single\_run}\ t = \exists\, s\, d\, i\, p.\ \text{is\_ftp}\ \ s\, d\, i\, p\, t \wedge \text{get\_port}\, s = 21 \\
\wedge\, \text{is\_in\_intranet}\, c \wedge \text{is\_in\_internet}\, s
\end{aligned} \tag{73}$$

where the predicates (is_in_intranet and is_in_internet) are used for checking if an address is within a specific network.

5.5. *Testing FTP*

The usual starting state for a test sequence is the empty trace and the given initial policy; the latter only allows ftp sessions (started using port 21, the control port of the ftp protocol) from the intranet to the internet:

$$\text{definition} \quad \sigma_0 = ([\,], \text{allow\_from\_to\_port}\, \text{intranet}\, \text{internet}\, \text{tcp}\, 21) \tag{74}$$

The keystone of the test section is the test specification:

$$\begin{aligned}
&\text{testspec test:} \\
&\qquad \text{length}\ (t) = \text{length}(X) \Longrightarrow \text{single\_run}\ (t) \Longrightarrow \\
&\qquad \sigma_0 \models os \leftarrow \text{mbind}\ t\, (\text{p2M policy});\ \text{return}(os\ =\ X) \\
&\Longrightarrow FUT\ \sigma_0\, t\ =\ X
\end{aligned} \tag{75}$$

The test case generation procedure searches for solutions for X and t that satisfy the first 3 constraints, namely that t must be a single protocol run, its execution on the FTP policy must yield a valid test sequence, and whose output must be equal to X. For solutions of this constraint resolution problem, a test may be run on FUT and checked if its behaviour conforms to X. This test specification is an instance of the test specification scheme discussed in Fact 58.

Using the presented test generation method and restricting the maximal length of test traces to four, four test cases are generated which each represent different FTP traces. The following is one example:

$$\begin{aligned}
&FUT\ \sigma_0\ [(1, (3, 9, \text{tcp}), (1005, 21\, \text{tcp}), \text{init}), (1, (3, 9\, \text{tcp}), (1005, 21\, \text{tcp}), \text{port\_request}\, 9), \\
&(1, (1005, 21\, \text{tcp}), (3, 9\, \text{tcp}), \text{data}), (1, (3, 9\, \text{tcp}), (1005, 21\, \text{tcp}), \text{close})] = \\
&\qquad\qquad\qquad\qquad [\text{allow}\ (), \text{allow}\ (), \text{allow}\ (), \text{allow}\ ()]
\end{aligned} \tag{76}$$

6. A DOMAIN-SPECIFIC TEST TOOL FOR FIREWALL POLICIES

So far, the presentation was concentrated on generating test data for abstract firewall policy models. In this section, the framework is extended to use this test data for testing real firewalls. Figure 3 illustrates the architecture of the framework for testing firewalls. This framework extends the HOL-TESTGEN workflow with components that are necessary for injecting the generated test data into a network that contains the firewall under test as well as components to monitor the test execution.

The *Test Execution* takes the generated test data as input and rewrites it as instructions that are then distributed to the Probes for execution. A description of expected results is also generated for use by the *Test Result Validation*. This rewriting steps also contain the input and output mappings inherently required in model-based testing: information needs to be provided how the inputs and outputs of the model are mapped to the real-world values and vice-versa (i. e. here, mainly how the addresses and the networks of the model are mapped to real IP addresses and networks).

The *Probes* have two basic roles: in the 'injector' role they send out packets according to the instructions, and in the 'sniffer' role they report packets they observe. These two basic roles are sufficient for testing stateless firewalls. Testing of stateful firewalls additionally requires a 'responder' role that can generate and send appropriate replies to packets belonging to certain protocols. In general, a probe may embody any combination of these three roles.

One of the main architectural units of the prototype is the *endpoint*. An endpoint is a combination of a network interface and a set of probes using it to sniff or insert packets. A single machine (virtual or real one) with multiple interfaces can host several endpoints. However an injector and a sniffer on the same interface count as one endpoint only—in most cases, each endpoint contains both. The endpoints should be connected as close to the firewall under test as possible (preferably with nothing else in between) and emulate the whole network which is (in normal operation) connected to the respective interface of the firewall under test.

The probes log events corresponding to the sending and sniffing of packets. This information is made available for evaluation by the *Test Result Validation*. The results analysis module takes as input files containing descriptions of the expected results (produced by the *Test Data Generation*, that is, HOL-TESTGEN), and the event logs recording the packets sent and received. It presents the results to the test engineer, allowing her to see where the actual results deviate from the expectations.

Recall the abstract test data from our stateless firewall example (Section 3.4):

1. $FUT(12, (2, 1, \text{tcp}), (1002, 80, \text{tcp}), \text{content}) = \llcorner\text{deny }()\lrcorner$
2. $FUT(8, (1002, 1, \text{udp}), (2, 8080, \text{udp}), \text{content}) = \llcorner\text{allow }()\lrcorner$

These abstract test data need to be refined into a format that can be injected into the network. In the framework, an adapted version of fwtest (http://user.cs.tu-berlin.de/~seb/fwtest/) is used. Thus, the

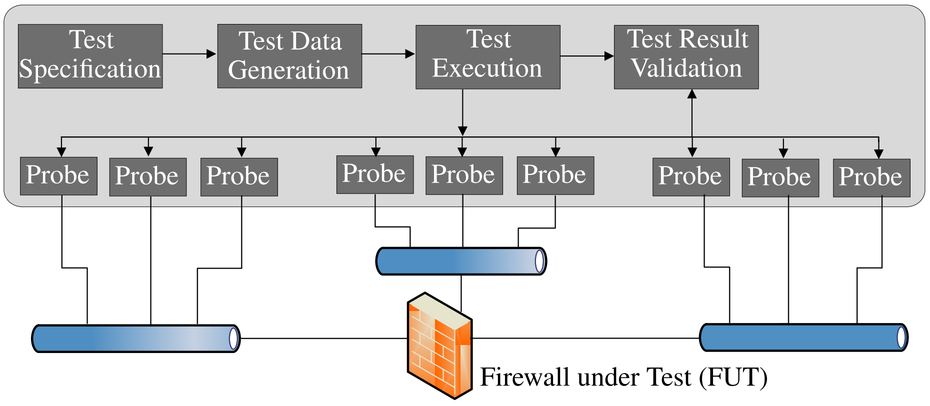


Figure 3. A framework for testing firewalls or routers.

abstract test data generated by HOL-TESTGEN have to be adapted to the input format of fwtest. For the given example, this gives:

```
62001:172.31.1.8:1:123.43.3.2:80:S:TCP:10
62002:123.43.3.2:1:172.31.1.8:8080::UDP:10
```

where the meaning of the individual parts (separated by a colon) is as follows: (unique) packet id, source IP, source port, destination IP, destination port, TCP flags, and the time to live. Moreover, also the expected test results need to be converted to a format used by fwtest:

```
62001 -- 1F -- 172.31.1.8:1:123.43.3.2:80:S:TCP -- deny
62002 -- 1F -- 123.43.3.2:12:172.31.1.8:8080::UDP:10 -- allow
```

where `1F` stands for 'filtered only, no NAT'

The following are the entries for a packet with NAT: the source port is mapped to a pool, that is, the source address is rewritten to a fixed new one with an arbitrary new source port number:

```
52041:10.11.1.8:1:10.11.2.1:67::UDP:10
52041 -- 1P -- 10.11.2.5:10.11.2.1:67:UDP -- allow
```

Similarly, the rewriting of IP addresses is supported automatically. For example, assume that the source IP is rewritten to a range of addresses:

```
52019:10.11.1.16:1:10.11.2.1:443:S:TCP:10
52019 -- SN -- 1:10.11.2.1:443:TCP:10.11.2.15:10.11.2.24 -- allow
```

7. EMPIRICAL RESULTS

In this section, empirical results of the test generation procedure are presented. For the evaluation, both real policies, for example drawn from the network of the ETH Zurich where part of this work emerged, as well as randomly generated policies, whose structure is inspired by ETH Zurich and industrial use cases, are used. In particular, randomly generated policies are used to estimate the correlation between the size of a policy and generation time of tests, to study the influence of various parameters of this correlation, as for example the address format used in the model, the number of involved networks and, of course, the impact of the normalization. The policies used were discussed with experts from industry to ensure that the evaluation fulfils the need of industrial partners from the telecommunication industry.

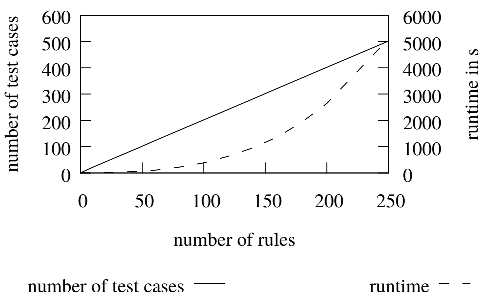
For lack of space, only stateless firewalls are evaluated in this presentation. Evaluations of stateful scenarios are discussed by [14].

Unless otherwise noted, all examples use an address representation consisting of a pair of Integers, the first denoting the host address, the second the port number. Furthermore, the indicated number of rules in the tables does not contain the default deny rule.

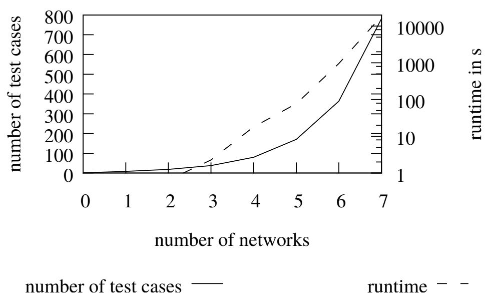
7.1. Packet filter with varying number of rules and networks

This scenario consists of what is often called a personal firewall. There are two networks: a workstation and the internet. The workstation consists of one single host, with all the remaining addresses belonging to the internet. All test specifications contain a constraint limiting the tests to packets that either have their source or their destination in the workstation network, and the other somewhere in the internet. The policies are constructed by a sequence of rules allowing packets arriving on one single port on the workstation, combined using an override operator and with a default deny rule in the end.

Figure 4a displays the number of generated test cases in relation to the number of rules allowing certain traffic in the policy n and the required time for test case generation. A relatively modest increase by adding a rule can be expected, as all the rules are conceptually very similar and only differ in one part of the input (although considering two others as well). This is reflected in the empirical results: the number of generated test cases for a policy consisting of n rules is $2n + 2$.



(a) The number of test cases and the runtime increase moderately when adding similar rules.



(b) The complexity of test case generation increases quickly when increasing the number of networks.

Figure 4. Policy complexity.

Table II. The number of test cases corresponds roughly to the number of rules, even if the policies are semantically very different.

Rules	10	20	30	40	50
Test cases	22	42	62	82	102
Test cases (summarized)	24	44	64	84	104

It is no surprise that two additional test cases are being generated for each new rule: there will be two additional packets with the new port number, one in each of the possible two directions. The required time increases slightly faster, but remains acceptable.

As an intermediate result from this experiment, it can be noted that policies of moderate complexity, in other words with similar rules, can be processed relatively efficiently even if consisting of a large number of rules.

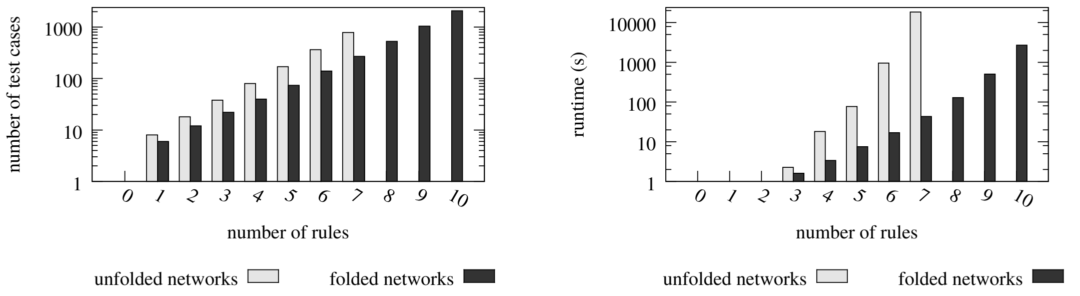
One important fact is that the processing time and the number of generated test data, and consequently also the achieved coverage, does not only depend on the semantics of the model of the policy, but also of its concrete representation. For example, a sequence of rules allowing one port between two networks might also be represented as a single rule allowing a set of port numbers between these networks. To explore the effect of such a representation on test case generation, the aforementioned experiment was adapted by always combining ten of the rules to one single rule (i. e. one rule allows a set of ten port numbers between the two networks) and run the algorithm again (Table II). It can be seen that the number of test cases is similar regarding the number of rules of the policy, even though in the second case the number of rules is ten times smaller for a semantically equivalent policy.

Next, the effect of varying another parameter is investigated: the number of networks. Again the policy ends with a default denial policy, which is prepended by a sequence of rules using an override operator. Each of the rules allows packets to the same port, originating in a new network and destined for the Internet. Each of the networks consists of a set of addresses.

Figure 4b illustrates the results. This time the number of generated test cases increases much more than in the previous case where the number of networks remained constant. As the time required also increases rapidly, only policies with a modest number of networks can be processed. There are two reasons that contribute to the fact that enlarging the number of networks increases complexity much larger than only enlarging the number of rules. First of all, there is the fact that the number of relevant input variables is now larger. Instead of only the port number, there are two additional variables for the networks (source and destination). Furthermore, those new input variables representing sets of sets of Integers are of a bigger internal complexity than the port numbers, which are modelled as simple Integers only.

At this stage it can already be noticed that the additional complexity in the rules leads test case generation to no longer scale using the naive approach without any normalization applied.

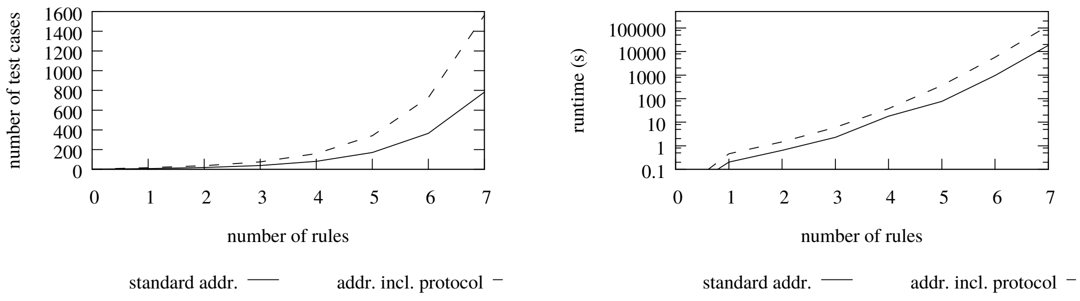
Figure 5a and Figure 5b present the outcome of generating tests for the same policies again with an increasing number of rules and networks, but where the network definitions have only



(a) The number of test cases decreases significantly when not unfolding the networks

(b) The runtime decreases significantly when not unfolding the networks.

Figure 5. Unfolding networks.



(a) The number of test cases increases when using a more complex address representation.

(b) The runtime increases even more when using a more complex address representation.

Figure 6. Comparing address representations.

been unfolded after application of the test case generation algorithm. A significant drop in both the number of generated test cases, as well as in the required time can be observed. As often the concrete values of the addresses do not matter, this approach is applicable in many cases. In essence, with this experiment the (partial) removal of the second effect on complexity mentioned before (internal complexity of the input variables) is simulated.

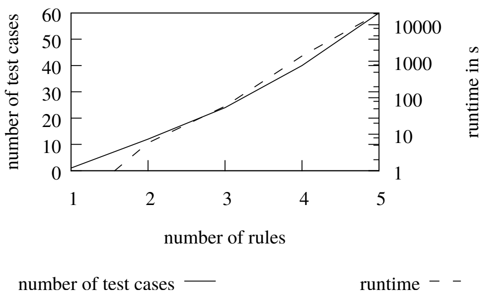
7.2. *Effect of address representation*

To study the effect of a more complex input type representation, the first scenario of the last example was also carried out with another address representation, including the protocol (as used in the examples in the previous sections). The rules of the policy are the same as before, but the packet protocol is additionally restricted to either tcp or udp (alternating). The effect is presented in Figure 6a and Figure 6b. It can be observed that the number of generated test cases doubles, corresponding to the two additional possibilities for each of the previous paths in the policy. Interestingly, the time complexity increases much faster.

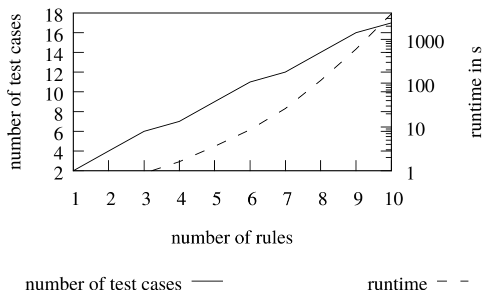
7.3. *Packet filter and NAT*

The next two examples evaluate the effects when testing a policy composed in parallel from two individual policies. More specifically, the effect of testing a combination of a NAT and a filtering policy is explored.

There are two networks in this scenario. The filtering policy consists of many rules allowing packets for differing ports from the first network to the second, and denying anything else. The NAT policy consists of many rules, each transforming certain packets originating from the first network. All other packets remain unchanged.

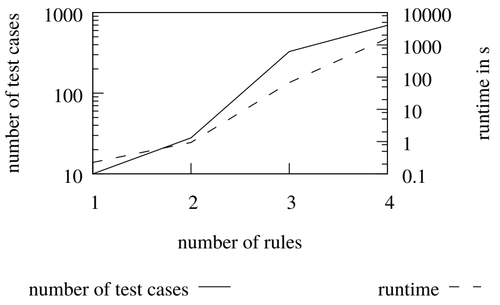


(a) The number of test cases increases quickly when adding new rules combined in parallel

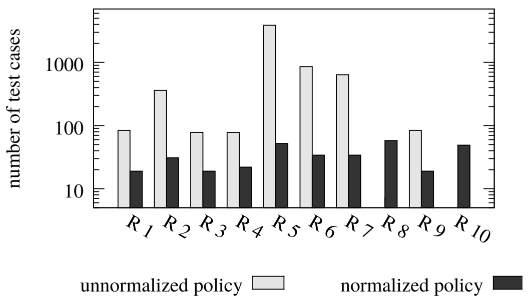


(b) The number of test cases increases less quickly when adding new rules to only one of the policies to be combined in parallel

Figure 7. Network address translation.



(a) The number of test cases increases substantially when the rules of the policy are not similar to each other.



(b) The normalization of policies decreases the number of test cases by several orders of magnitude.

Figure 8. Non-uniform policies and normalization.

The example is carried out in two variants: in the first one, the number of rules of both sub-policies increases uniformly (Figure 7a). In the second variant, the number of filtering rules is fixed to three and only the number of NAT rules increases (Figure 7b). It can be observed that when increasing the number of rules in both policies in a rather non-related way, the complexity increases substantially.

7.4. *A non-uniform policy*

So far, diverse scenarios have been considered where one or more parameters were changed in a controlled manner. The next example is an experiment with a small non-uniform policy with very diverse rules. The choice of the rules is random and thus the results of the experiment are of limited value. Furthermore, most real world policies are rather uniform. Nonetheless, this experiment gives an indication about the effects of processing non-standard policies.

The policy consists of a default deny rule, to which sequentially the following rules are added using an override operator.

- AllowPortTo *subnet1 internet* 80
- DenyFromTo *subnet2 subnet4*
- AllowPortRangeTo *subnet3 subnet1* $\{80, 180, 280\}$
- AllowFromTo *subnet3 subnet4*

Explicitly no coverage- and complexity-reducing strategy are employed. In particular all concepts are unfolded for the test case generation algorithm (Figure 8a).

This small example has confirmed that adding diverse rules and processing them in a non-optimal manner will lead to a very large if-then-else cascade and consequently a very high complexity of

Table III. Test case generation for firewall policies utilising normalization.

		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	E1	E2
Original	Networks	2	3	4	3	4	3	4	5	5	6	4	5
	Rules	13	9	5	9	13	13	8	15	5	11	11	8
	TC Gen. (s)	4.8	38	2.4	2.8	10216	309	119	—	2.8	—	42	4
	Test Cases	84	360	78	78	3864	860	637	—	84	—	322	130
Normalized	Rules	8	14	11	10	24	15	17	28	11	25	17	16
	Segments	2	4	5	3	7	4	6	9	5	9	6	5
	TC Gen. (s)	0.5	1.4	0.5	0.8	2.3	1.6	1.4	2.5	0.6	1.9	1.0	1.0
	Test Cases	19	31	19	22	52	34	34	58	19	49	32	34

the test case generation. Furthermore, the relative increase in both the time required and the number of generated test cases when adding one single rule largely depends on the kind of the rule.

7.5. *Normalization*

In a first experiment, the impact of policy normalization was explored on the overall time required for generating test cases. Table III summaries these results for randomly generated policies (R1 to R10) as well as policies used at ETH Zurich (E1 and E2). The most important result is that while the number of rules can increase during normalization, the time required for generating test cases is significantly smaller after normalization, even if the time needed for normalization is included. While for some policies, the test case generation took more than 24 hours before normalization, the required time for normalizing the policy and generating test cases for the normalized policy is only a few seconds. The increase in the number of rules can be explained by the fact that during normalization, segment-specific deny rules need to be inserted. The significant reduction of test case generation time can be explained by the fact that after segmentation, each segment specific policy is much smaller and less complex. As the normalization of policies reduces the number of test cases significantly (Figure 8b), the time needed for executing a test on an implementation is also reduced substantially.

The generation of test cases for larger policies in reasonable time is only possible after normalization. Thus, the effect of the normalization was investigated in isolation. Table IV shows the normalization results for randomly generated policies with different sizes both in the number of rules and the number of networks covered (see also Figure 9a and Figure 9b). All policies end with a DenyAll rule and one in every five rules is a DenyAllFromTo. If the number of rules is small related to the number of networks, the number of rules after normalization increases quite significantly as there are a lot of additional DenyAllFromTo's to be inserted. In the opposite case, it decreases a lot, as many of the AllowPortFromTo rules will be removed during normalization. The time needed for the normalization primarily depends upon the number of networks and on the number of rules.

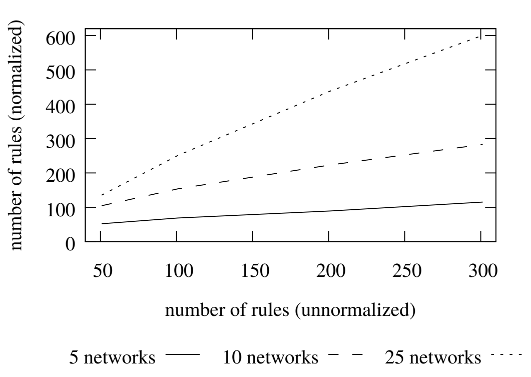
To shed some light on the influence of algorithmic factors of the transformation, this series of experiments has been carried out with different sorting procedures, namely insertion sort and quicksort. Quite large differences can be observed, especially with the large-scale policies. It must be noted that the sorting phase is by far the most time-consuming one.

The table also presents the number of test cases generated by applying the test case generation algorithm to the normalized policies. The time required remains very low compared with the time used for normalization. Even for the biggest policy, it requires less than three minutes for both test case and test data generation combined.

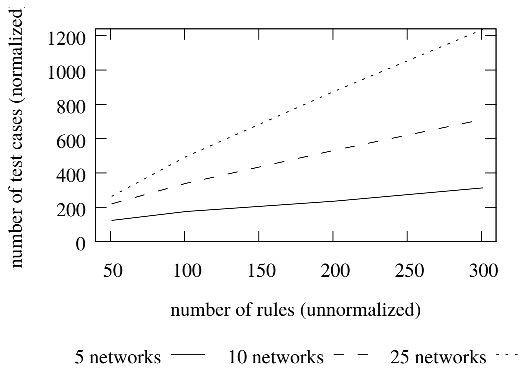
As a last measurement, to present the effect of normalizing parallel composed policies, a couple of policies have been defined as parallel composition of a filtering and a NAT policy. The normalization is transferred over the composition operator, as outlined in Section 4.5. The filtering rules are defined to allow each a distinct port between two distinct networks. For each filtering rule, there is a corresponding NAT rule performing source address translation to a pool of addresses, rewriting all packets having the source in the same network as the source of the corresponding filtering rule. Each policy of the experiment is constructed by an override composition of both policies individually (including a default rule for each), and performing the parallel composition in the end

Table IV. The size of a normalized policy mainly depends on the number of rules in relation to the number of networks.

Rules	51	51	51	101	101	101	201	201	201	301	301	301
Networks	5	10	25	5	10	25	5	10	25	5	10	25
Segments	11	32	48	11	41	86	11	46	147	11	46	188
Rules	52	105	136	69	154	251	89	223	438	115	283	600
Average	5	3	3	7	4	3	9	5	3	11	6	3
Time[s] (insort)	33	83	67	102	373	590	212	1731	6460	516	2782	34248
Time[s] (qsort)	31	112	522	91	318	1593	206	765	7960	453	1198	17355
Test cases	124	220	265	175	340	496	235	532	877	313	713	1237

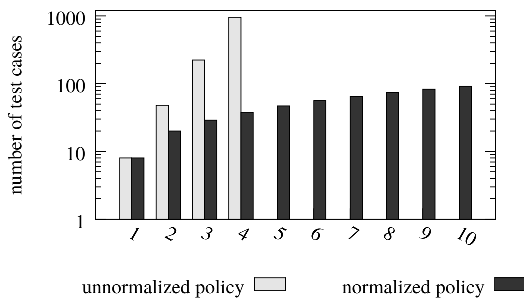


(a) The size (number of rules) of a policy after normalization increases with both the number of rules in the normalized policy and the number of networks.

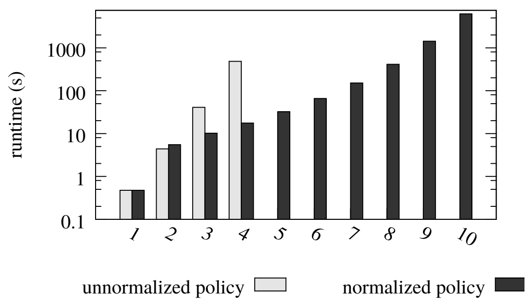


(b) The number of test cases of a policy after normalization increases with both the number of rules in the normalized policy and the number of networks.

Figure 9. Policy normalization.



(a) The number of test cases decreases significantly for combined NAT and filtering policies (logarithmic scale)



(b) The runtime decreases significantly for combined NAT and filtering policies (logarithmic scale)

Figure 10. Combining NAT and filtering policies.

only. Recalling the previous section, naive test case generation for such a policy would not scale. For this reason, the normalization procedure for filtering policies presented before, which distributes over to the combined policy, is applied. The results of applying the normalization on these policies are presented in Figure 10a, Figure 10b and Table V. The number of rules is the number for each sub-policy. It can be observed that large gains in efficiency are achieved. These could even be increased if a transformation procedure would be applied also on the NAT part of the policy.

As a summary of the last couple of experiments, the application of policy transformation techniques allows for a relatively efficient testing technology for a wide range of large-scale policies as encountered in practice. Of course, the coverage regarding the original specification is reduced, and consequently also the probability of detecting failures. However, mostly only those parts from the specification have been removed from being tested that are very unlikely to contain any failures, by splitting the policy into independent parts. The experiments provided evidence that in practice

Table V. The number of test cases and the runtime of parallel composed policies: application of transformation procedures increases efficiency substantially.

Rules	1	2	3	4	5	6	7	8	9	10
Test cases (normalized)	8	20	29	38	47	56	65	74	83	92
Test cases (original)	8	48	224	960	-	-	-	-	-	-
Runtime (normalized)	1	5	10	18	32	66	151	414	1427	6225
Runtime (original)	1	4	41	483	-	-	-	-	-	-

the ability to detect failures is not reduced significantly, and overall effectiveness of the testing procedure has been improved substantially.

8. CONCLUSION AND RELATED WORK

8.1. Related work

While several approaches for the formal specification, e. g., [15, 23, 33, 39], as well as specification-based testing, e. g., [20, 21, 28, 30], of firewalls have been proposed, none of them supports NAT or uses transformations of policies for increasing the effectiveness of the test case generation. Given the large address space of common network protocols (e. g. 2^{32} addresses for IPv4 and 2^{128} addresses for IPv6) there is a common understanding that random testing approaches are not feasible.

All existing formal specification approaches model the decision of a firewall as mapping to a simple binary domain (e. g., `allow` or `deny`) and, thus, do not support NAT. For example, Capretta et al. [15] present a formalization of network addresses in Coq [3] that is quite close to the presented model for stateless packet filters discussed in Section 3.2 using records. The authors model security policies as set of rules (instead of partial functions). Thus, they are interested in detecting conflicting rules, that is, rules that map the same network packets to different decisions. For this, they present a conflict detection algorithm together with a formal correctness proof. Gawanmeh and Tahar [23] achieve a similar result using Event-B [1]. Using a similar model for stateless packet filters, Matsumoto and Bouhoula [33] discuss a mapping to Boolean formulae and use a SAT solver for implementing a tool that verifies the correctness of stateless firewall configurations with respect to a given specification. A similar result, using an SMT solver, is presented by Youssef and Bouhoula [39] for stateful firewalls. Their model of application-level stateful firewalls does, similar to the model presented in this paper, discuss a deterministic FTP variant as an example.

Jürjens and Wimmel [30] propose specification-based testing of firewalls that employs a finite formal model of the network, modelling all entities as simple datatypes. This model is used for deriving test cases automatically. Unfortunately, the test case generation technique is not really described, and the tested configurations appear small. Apparently, it does not apply any policy transformation technique. Bishop et al. [4] describe a formal model of protocols in HOL. However, their level of abstraction is much lower than ours; therefore, it is much less suited for testing of policy conformance. Marmorstein and Kearns [32] propose a policy-based host classification which can be used to detect errors and anomalies in a firewall policy. Finally, Hwang et al. [28] present an informal approach to firewall conformance testing that generates test cases based on certain structural coverage criteria (e. g. rule coverage).

El-Atawy et al. [20, 21] present a purely syntactic approach for the segmentation of policies into sub-policies that restrict the traffic between pairs of networks. Overall, this result of their segmentation is quite similar to the presented policy normalization, which also groups policy rules according to the networks. While this is the only work that discusses a transformation of firewall policies in the context of test case generation, transforming security policies to improve their overall runtime performance is a well-known technique. Liu et al. [31] present algorithms based on decision diagrams, that minimize the number of rules of a firewall policy. We are not aware of any attempt to formally verify the correctness of the policy transformation technique.

Dssouli et al. [19] have already introduced a notion of testability for communication protocols using a specification based on finite state machines. They present a method for designing communication protocols that are ‘easy' to test. The closest related work, however, is that of Harman et al. [25], which is further developed in, e. g., Harman [24], Harman et al. [26], Hierons et al. [27]. They introduce the concept of *testability transformations*, that is, source-to-source transformations of programmes that increase their testability. Similar to work presented in this paper, the goal of these transformations is minimization of the test cases required to achieve full test coverage (with respect to a given test adequacy criterion). While their work is based on transformations of the source code, that is, the system under test, we transform a formal specification, which makes our approach applicable in black-box testing scenarios. Furthermore, in contrast to their work, our transformation procedures allow for explicit modifications of the test adequacy criterion with respect to the original form of the specification, something which we believe to be necessary for being able to realistically perform conformance testing of large-scale security policies.

In contrast to these works, we provide a uniform, tool-supported approach for formally specifying and analysing firewall specifications, transforming a test specification to improve the testability, and generating test cases for an implementation. Thus, we provide an efficient and effective approach for testing the conformance of a firewall (and its configuration) to a specification (i. e. high level policy) that supports advanced concepts such as network address (or port) translation. Moreover, the presented approach integrates the testing of both stateless and stateful protocols. And last but not least, our approach is based on a formal proof that the applied testability transformation preserves semantics.

8.2. *Conclusion and future work*

In this paper, a formal model of stateless and stateful firewall policies in Isabelle/HOL as an instantiation of the UPF was presented. The formal model is supported by a policy normalization approach that is formally proven correct and which increases the effectiveness of the specification-based test case generation approach by several orders of magnitude. We integrated modelling, derived rules for symbolic computation, test generation and adapted test execution in a framework that allows a seamless transition from particularly clean semantic foundations to a practically useful and competitive application.

Harman [24] discusses a list of open problems in testability transformations. With this work, a significant contribution to the final problem in his list is provided: testability transformations for specification-based testing. In particular, the policy normalization acts as testability translation for the model-based generation of test cases for firewalls. Moreover, the correctness, that is, semantics preservation of transformations was proven formally within the same framework used for test case generation.

With HOL-TESTGEN's firewall support, we believe the first test tool to integrate the specification, formal analysis, transformation, test case generation, and test execution of firewall policies including support for NAT as well as stateful protocols has been developed.

Besides applying test case generation to other kinds of security domains, for example, [5], we see several productive lines of future work. On the theoretical side, developing formal testability transformations for sequence testing seems to be particularly appealing. This work would address the second to last problem of Harman [24]: testability transformations for specifications based on finite-state machines. Moreover, considering testability transformations that do not preserve the semantics of the specifications, as suggested by Harman et al. [26], requires the development of new test hypotheses. Here, the integrated approach of HOL-TESTGEN allows for the formal computation and analysis of such hypotheses (cf. Brucker et al. [9]). Finally, the relation between testability transformation and fault models needs to be investigated further.

APPENDIX A: CORE PART OF THE CORRECTNESS PROOF OF THE RULE SORTING ALGORITHM

This theorem and its formal proof script is crucial for proving the correctness of the sorting algorithm as described in Section 4.4. It states that for any two lists consisting of the same rules and conforming to two wellformedness conditions, the semantics is the same under the assumption that all networks are either equal or pairwise distinct. The proof is performed by doing a case distinction

```
theorem C_eq_Sets_mr:
 assumes sets_eq: "set p = set s" and SC: "singleComb p" and wp1_p: "wellformed_policy1_new p"
     and wp1_s: "wellformed_policy1_new s" and wp3_p: "wellformed_policy3 p"
     and wp3_s: "wellformed_policy3 s" and aND: " allNetsDistinct p"
 shows "matching_rule x p = matching_rule x s"
proof (cases "matching_rule x p")
 case None
     have DA: "DenyAll ∈ set p" using wp1_p by (auto simp: wp1_aux1aa)
     have notDA: "DenyAll ∉ set p" using None by (auto simp: DAimplieMR)
     thus ?thesis using DA by (contradiction)
 next
 case (Some y) thus ?thesis
   proof (cases y)
     have tl_p: "p = DenyAll#(tl p)" by (metis wp1_p wp1n_tl)
     have tl_s: "s = DenyAll#(tl s)" by (metis wp1_s wp1n_tl)
     have tl_eq: "set (tl p) = set (tl s)" by(metis tl.simps(2) WP1n_DA_notinSetfoo2
                    mem_def sets_eq wellformed_policy1_charn wp1_aux1aa wp1_eq wp1_p wp1_s)
{    case DenyAll
     have mr_p_is_DenyAll: "matching_rule x p = Some DenyAll" by (simp add: DenyAll Some)
     hence x_notin_tl_p: "∀ r. r ∈ set (tl p) ⟶ x ∉ dom (C r)"
         using wp1_p by (auto simp: mrDenyAll_is_unique)
     hence x_notin_tl_s: "∀ r. r ∈ set (tl s) ⟶ x ∉ dom (C r)"
         using tl_eq by auto
     hence mr_s_is_DenyAll: "matching_rule x s = Some DenyAll" using tl_s
          by (auto simp: mr_first)
     thus ?thesis using mr_p_is_DenyAll by simp
}{   case (DenyFromTo a b)
     have mr_p_is_DAFT: "matching_rule x p = Some (DenyFromTo a b)"
          by (simp add: DenyFromTo Some)
     have DA_notin_tl: "DenyAll ∉ set (tl p)" by (metis WP1n_DA_notinSet wp1_p)
     have mr_tl_p: "matching_rule x p = matching_rule x (tl p)"
       by (metis Comb.simps(1) DenyFromTo Some mrConcEnd tl_p)
     have dom_tl_p: "⋀ r. r ∈ set (tl p) ∧ x ∈ dom (C r) ⟹ r = (DenyFromTo a b)"
       using wp1_p aND SC wp3_p mr_p_is_DAFT by (auto simp: rule_charnDAFT)
     hence dom_tl_s: "⋀ r. r ∈ set (tl s) ∧ x ∈ dom (C r)
                         ⟹ r = (DenyFromTo a b)" using tl_eq  by auto
     have DAFT_in_tl_s: "DenyFromTo a b ∈set (tl s)"
       using mr_tl_p by (metis DenyFromTo mrSet mr_p_is_DAFT tl_eq)
     have x_in_dom_DAFT: "x ∈dom (C (DenyFromTo a b))"
       by (metis mr_p_is_DAFT DenyFromTo mr_in_dom)
     hence mr_tl_s_is_DAFT: "matching_rule x (tl s) = Some (DenyFromTo a b)"
       using DAFT_in_tl_s dom_tl_s by (auto simp: mr_charn)
     hence mr_s_is_DAFT: "matching_rule x s = Some (DenyFromTo a b)"
       using tl_s
       by (metis  DA_notin_tl DenyFromTo EX_MR mrDA_tl mr_p_is_DAFT not_Some_eq tl_eq
                  wellformed_policy1_new.simps(2))
     thus ?thesis using mr_p_is_DAFT by simp
}{   case (AllowPort a b c)
     have wp1s: "wellformed_policy1 s" by (metis wp1_eq wp1_s)
     have mr_p_is_A: "matching_rule x p = Some (AllowPort a b c)"
                     by (simp add: AllowPort Some)
        hence A_in_s: "AllowPort a b c ∈ set s" using sets_eq by (auto intro: mrSet)
     have x_in_dom_A: "x ∈dom (C (AllowPort a b c))" by (metis mr_p_is_A AllowPort mr_in_dom)
     have SCs: "singleComb s" using SC sets_eq by (auto intro: SCSubset)
     hence ANDs: " allNetsDistinct s" using aND sets_eq SC by (auto intro: aNDSetsEq)
     hence mr_s_is_A: "matching_rule x s = Some (AllowPort a b c)"
          using A_in_s wp1s mr_p_is_A aND SCs wp3_s x_in_dom_A by (simp add: rule_charn2)
     thus ?thesis using mr_p_is_A by simp
}    case (Conc a b) thus ?thesis by (metis Some mr_not_Conc SC)
 qed
qed
```

of the rule that matches a packet x in the first list. There are four cases: for the DenyAll operator, it can be shown that there cannot be any other rule in the lists that matches. The $(a \oplus_S b)$ case can be ruled out by contradiction. In the other two cases, it can be shown that the matching rule of the second list must necessarily be the same one.

ACKNOWLEDGEMENTS

This work was partially funded by BT Group plc and by the Digiteo Foundation.

REFERENCES

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, New York, NY, USA, 2009.

[2] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, Heidelberg, 2004.

[4] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 55–66. ACM Press, 2006.

[5] A. D. Brucker and B. Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology (STTT)*, 7(3):233–247, 2005. doi: 10.1007/s10009-004-0176-3.

[6] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007. doi: 10.1007/978-3-540-73770-4_9.

[7] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing (FAC)*, 25(5):683–721, 2013. doi: 10.1007/s00165-012-0222-y.

[8] A. D. Brucker, L. Brügger, and B. Wolff. Model-based firewall conformance testing. In K. Suzuki and T. Higashino, editors, *Testcom/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, 2008. doi: 10.1007/978-3-540-68524-1_9.

[9] A. D. Brucker, L. Brügger, and B. Wolff. Verifying test-hypotheses: An experiment in test and proof. *Electronic Notes in Theoretical Computer Science*, 220(1):15–27, 2008. doi: 10.1016/j.entcs.2008.11.003. Proceedings of the Fourth Workshop on Model Based Testing (MBT 2008).

[10] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test-case generation. In *Third International Conference on Software Testing, Verification, and Validation (ICST)*, pages 345–354. IEEE Computer Society, 2010. doi: 10.1109/ICST.2010.50.

[11] A. D. Brucker, L. Brügger, M. P. Krieger, and B. Wolff. HOL-TestGen 1.5.0 user guide. Technical Report 670, ETH Zurich, Apr. 2010.

[12] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. An approach to modular and testable security models of real-world health-care applications. In *ACM symposium on access control models and technologies (SACMAT)*, pages 133–142. ACM Press, 2011. doi: 10.1145/1998441.1998461.

[13] A. D. Brucker, L. Brügger, and B. Wolff. HOL-TestGen/FW: An environment for specification-based firewall conformance testing. In Z. Liu, J. Woodcock, and H. Zhu, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, number 8049 in Lecture Notes in Computer Science, pages 112–121. Springer-Verlag, 2013. doi: 10.1007/978-3-642-39718-9_7.

[14] L. Brügger. *A Framework for Modelling and Testing of Security Policies*. PhD thesis, ETH Zurich, 2012. ETH Dissertation No. 20513.

[15] V. Capretta, B. Stepien, A. Felty, and S. Matwin. Formal correctness of conflict detection for firewalls. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, FMSE '07, pages 22–30, New York, NY, USA, 2007. ACM. doi: 10.1145/1314436.1314440.

[16] Center for Strategic and International Studies. Securing cyberspace for the 44th presidency, Dec. 2008.

[17] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[18] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Heidelberg, 2008. Springer-Verlag. doi: 10.1007/978-3-540-78800-3_24.

[19] R. Dssouli, K. Karoui, K. Saleh, and O. Cherkaoui. Communications software design for testability: specification transformations and testability measures. *Information and Software Technology*, 41(11-12):729–743, 1999. doi: 10.1016/S0950-5849(99)00033-6.

[20] A. El-Atawy, K. Ibrahim, H. Hamed, and E. Al-Shaer. Policy segmentation for intelligent firewall testing. In *NPSec 05*, pages 67–72. IEEE Computer Society, Nov. 2005.

[21] A. El-Atawy, T. Samak, Z. Wali, E. Al-Shaer, F. Lin, C. Pham, and S. Li. An automated framework for validating firewall policy enforcement. In *POLICY '07*, pages 151–160. IEEE Computer Society, 2007.

[22] M. C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, TAPSOFT*'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 82–96. Springer-Verlag, Heidelberg, 1995.

[23] A. Gawanmeh and S. Tahar. Domain restriction based fromal model for firewall configurations. *International Journal for Information Security Research (IJISR)*, 2, March/June 2012.

[24] M. Harman. Open problems in testability transformation. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 196–209, April 2008. doi: 10.1109/ICSTW.2008.30.

[25] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Trans. Softw. Eng.*, 30(1):3–16, 2004. doi: 10.1109/TSE.2004.1265732.

[26] M. Harman, A. Baresel, D. Binkley, R. M. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. Testability transformation - program transformation to improve testability. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 320–344, Heidelberg, 2008. Springer-Verlag. doi: 10.1007/978-3-540-78917-8_11.

[27] R. M. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *Comput. J.*, 48(4):421–436, 2005. doi: 10.1093/comjnl/bxh093.

[28] J. Hwang, T. Xie, F. Chen, and A. Liu. Systematic structural testing of firewall policies. *Network and Service Management, IEEE Transactions on*, 9(1):1–11, march 2012. doi: 10.1109/TNSM.2012.012012.100092.

[29] International Organization for Standardization. ISO/IEC DIS 29119: Software and Systems Engineering—Software Testing. ISO Draft International Standard, July 2012.

[30] J. Jürjens and G. Wimmel. Specification-based testing of firewalls. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*, pages 308–316. Springer-Verlag, 2001.

[31] A. X. Liu, E. Torng, and C. Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *Proceedings of the the 27th Annual IEEE Conference on Computer Communications (Infocom)*, Phoenix, Arizona, April 2008.

[32] R. Marmorstein and P. Kearns. Firewall analysis with policy-based host classification. In *Large Installation System Administration Conference*, pages 4–4. USENIX Association, 2006.

[33] S. Matsumoto and A. Bouhoula. Automatic verification of firewall configuration with respect to security policy requirements. In *Proceedings of the International Workshop on Computational Intelligence in Security for Information Systems (CISIS)*, pages 123–130, Heidelberg, 2008. Springer-Verlag. doi: 10.1007/978-3-540-88181-0_16.

[34] 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-Verlag, Heidelberg, 2002. doi: 10.1007/3-540-45949-9.

[35] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY USA, 1993. ACM Press. doi: 10.1145/158511.158524.

[36] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Internet Standard), Oct. 1985.

[37] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, 1999. doi: 10.1145/300830.300839.

[38] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1992.

[39] N. B. Youssef and A. Bouhoula. Dealing with stateful firewall checking. In H. Cherifi, J. M. Zain, and E. El-Qawasmeh, editors, *DICTAP (1)*, volume 166 of *Communications in Computer and Information Science*, pages 493–507, Heidelberg, 2011. Springer-Verlag. doi: 10.1007/978-3-642-21984-9_42.