Verified Firewall Policy Transformations for Test Case Generation

Achim D. Brucker*, Lukas Brügger†, Paul Kearney‡, and Burkhart Wolff§
*SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
Email: achim.brucker@sap.com
† Information Security, ETH Zurich, 8092 Zurich, Switzerland[1]
Email: lukas.bruegger@inf.ethz.ch
‡Security Futures Practice, BT Innovate & Design, Adastral Park, Ipswich, UK
Email: paul.3.kearney@bt.com
§Université Paris-Sud, Parc Club Orsay Université, 91893 Orsay Cedex, France[2]
Email: wolff@lri.fr

***Abstract*—We present an optimization technique for model-based generation of test cases for firewalls. Starting from a formal model for firewall policies in higher-order logic, we derive 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. The correctness of the rules and the algorithm is established by formal proofs in Isabelle/HOL. Finally, we use the normalized policies to generate test cases with the domain-specific firewall testing tool HOL-TESTGEN/FW.**

The resulting procedure is characterized by a gain in efficiency of two orders of magnitude. It can handle configurations with hundreds of rules such as frequently occur in practice.

Our approach can be seen as an instance of a methodology to tame inherent state-space explosions in test case generation for security policies.

***Keywords*-Security testing, model-based testing**

I. Introduction

Firewalls suffer from the same quality problems as other complex software, but mature products from established and trusted vendors are generally trustworthy and any vulnerabilities tend to be found and patched relatively quickly. Given this situation, is there need for anyone other than firewall manufacturers, independent labs, and organizations with critical security requirements to test firewalls? The answer is "yes," because the off-the-shelf product must be configured with a rule set that implements an appropriate security policy to create a working firewall. The likelihood of a security vulnerability arising from misconfiguration is much greater than from a bug the software itself:

> "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." [1, p. 55]

As firewalls tend to have complicated configurations, this seems to be particularly true for them. Furthermore, firewall policies and hence rule sets change with time. Often, changes are implemented by adding rules, resulting in ever-growing complexity, which increases the probability of errors and the challenge of finding them. Thus, there is a continuing need to re-validate that the configured firewall complies with the security policy, and the importance and difficulty of this increases with time. While it is useful to verify the rule set by inspection helped by such analysis tools as are available, this is no substitute for actually testing that the firewall really behaves as specified by the security policies from which the rule set is derived.

To appreciate the potential consequences of errors in apparently trustworthy network components, consider this recent incident in which an inexperienced administrator coupled with an intolerance of certain widely used routers to long autonomous system paths, led to a partial Internet break-down:

> "On Monday, February 16, 2009, a misconfigured router from SuproNet, a Czech Internet Service Provider, caused high increases in Border Gateway Protocol (BGP) updates as well as isolated outages for Internet services around the world." [2]

In [3], we presented a model-based testing technique for network components such as routers or firewalls. Based on a formal model of networks, packets, and security policies, our HOL-TESTGEN system automatically generates test-sets covering all the partitions of the underlying disjunctive normal form (DNF), at least for small policies. In this paper, we concentrate on improving the effectiveness of our approach to make it more feasible for larger applications. Even modest size companies will have firewalls with several hundred rules. While it is possible that these rule sets could be partitioned into largely independent "virtual firewalls", scalability is an important property for any practically-relevant test-generation technique. Constructing systematic tests on security policies leads inevitably to large cascades of case distinctions over input and output. The situation is worse if the underlying state is evolving over time (as in stateful firewalls).

A careful analysis of the constraint resolution process of HOL-TESTGEN loaded with the firewall policy theory de-

[1]This work was supported by BT Group plc.
[2]This work was partially supported by the Digiteo Foundation.

scribed in [3] revealed that many case-splits are concerned with intra-subnet communication and are in fact redundant for the overall problem of communication between networks across a firewall. Since redundant cases can already be detected syntactically on a sequence of policy rules, this gives rise to the idea of a normalization of policies before the actual case-splitting is executed.

Our contributions are three-fold: first, we present a firewall policy transformation that: a) eliminates redundant (shadowed) rules, b) groups rules along the subnet-hierarchy, and c) produces tests only between pairs of networks. Second, we show the correctness of our firewall policy transformation formally using Isabelle/HOL, and define a function in higher-order logic (also shown to be correct) that applies the rules to achieve a normal form. And, third, we show in several case studies, that this policy transformation increases the efficiency of test case generation by at least two orders of magnitude.

The rest of the paper is structured as follows: After an introduction to the HOL-TESTGEN approach (Section II) and our formal firewall model (Section III), we present a number of policy transformations in Section IV and a normalization procedure in Section V. Both sections include a formal proof of correctness. The paper concludes with an empirical evaluation and a general discussion of the methodology.

II. FORMAL AND TECHNICAL BACKGROUND

A. Isabelle and Higher-order Logic

Isabelle [4] 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 Higher-order logic (HOL) [5, 6].

HOL is a classical logic with equality, based on typed total higher-order functions; the type discipline rules out paradoxes such as Russel's paradox in untyped set theory. As a term language, HOL is based on the typed λ-calculus—i. e. the *terms* of HOL are λ-expressions. Types of terms may be built from *type variables* (like α, β, ...) or *type constructors* (like bool or nat). Type constructors may have arguments (as in $\alpha\ \mathrm{list}$ or $\alpha\,\mathrm{set}$). The type constructor for the function space, $\Rightarrow$, is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ have the alternative syntax $[\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}$. HOL is centered around the extensional logical equality, $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow \mathrm{bool}$, where bool is the fundamental logical type. We use infix notation: instead of $(\_ = \_)\ E_1\ E_2$ we write $E_1 = E_2$. The logical connectives $\_ \wedge \_$, $\_ \vee \_$, $\_ \rightarrow \_$ of HOL have type $[\mathrm{bool}, \mathrm{bool}] \Rightarrow \mathrm{bool}$, $\neg\_$ has type $\mathrm{bool} \Rightarrow \mathrm{bool}$. The quantifiers $\forall\,\_.\_$ and $\exists\,\_.\_$ have type $[\alpha \Rightarrow \mathrm{bool}] \Rightarrow \mathrm{bool}$; thus, quantifiers may range over types of higher order, i. e. functions. This core language can easily be extended in a logically-safe way by Cartesian products (written (a, b) of type $\alpha \times \beta$), sets (with operators such as membership $\_ \in \_ :: [\alpha, \alpha\ \mathrm{set}] \Rightarrow \mathrm{bool}$, the set comprehension $\{\_.\_\} :: (\alpha \Rightarrow \mathrm{bool}) \Rightarrow \alpha\ \mathrm{set}$, $\_ \cup \_$, $\_ \cap \_ :: [\alpha\ \mathrm{set}, \alpha\ \mathrm{set}] \Rightarrow \alpha\ \mathrm{set}$) and lists (with operators such as *hd* and *tl* and the concatenation $\_@\_ :: [\alpha\ \mathrm{list}, \alpha\ \mathrm{list}] \Rightarrow \alpha\ \mathrm{list}$). The HOL library comprises a theory of maps to model partial functions. They are written as $\alpha \rightharpoonup \beta$, which is a type synonym for $\alpha \Rightarrow \beta\ \mathrm{option}$, where $\beta\ \mathrm{option}$ is a datatype consisting of the two elements None and $\mathrm{Some}\,\beta$. Over those, the usual concepts of override $\_ {+\!\!+} \_ :: [\alpha \rightharpoonup \beta, \alpha \rightharpoonup \beta] \Rightarrow \alpha \rightharpoonup \beta$, domain $\mathrm{dom}\ f$ and range $\mathrm{ran}\ f$ are introduced.

B. The HOL-TestGen System

HOL-TESTGEN is an interactive, i. e. semi-automated, test tool for specification-based tests built upon Isabelle/HOL. We briefly review main concepts and outline the standard workflow. The latter is divided into five phases: 1) writing the *test theory*, i. e. a collection of basic types and auxiliary functions formalizing the problem domain, 2) writing the *test specification*, TS, specifying the concrete property to be tested, 3) the *test case generation* phase, i. e. an automated conversion of TS into a sequence of *test cases*, TC, (or: partitions) representing classes of possible input, 4) the *test data generation* phase, where concrete members are constructed for the TC, and 5) the *test execution* phase where HOL-TESTGEN generates a *test script* driving the actual testing. Once a test theory is completed, documents can be generated that represent a formal test plan, including test theory, test specifications, configurations of the test data and test script generation commands. The plan may also include proofs for rules that support the overall process and can be processed either in batch mode or interactively, and optionally the results of the test execution.

The core of the test case generation procedure lies in case splittings up to a certain depth for each free or universally quantified (input) variable in the test specification; depth and form of the case split depend on the type of the variable. The resulting test cases, TC_i, have the form $C_1\ x \wedge \cdots \wedge C_n\ x \rightarrow P(put\ x)$, where put is a place-holder for the program under test, x the input vector and P the oracle or postcondition telling that the output of put complies to the test specification. Test data generation from test cases boils down to a constraint resolution process finding an x satisfying the constraints C_i. The reader interested in more details over theory and implementation is referred to [7, 8].

III. MODELING FIREWALLS IN HOL REVISITED

In this section, we introduce our model of firewall policies and present a formal theory for them. For space limitations, however, we will restrict ourselves to the case of stateless firewalls, also called *stateless packet filters*. The model of stateful firewall policies is presented in [8].

A. A Formal Firewall Model

A message from network A to network B is split into several *packets* that contain 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 filters traffic passing from one network to another based on that routing information and the *policy* used. The policy describes which packets should be accepted and which should be rejected.

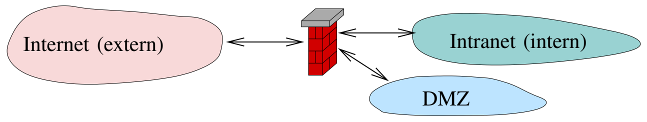

Figure 1. A simple firewalling scenario.

source	destination	protocol	port	action
Internet	DMZ	smtp	25	accept
Internet	DMZ	http	80	accept
intranet	DMZ	smtp	25	accept
DMZ	intranet	smtp	25	accept
intranet	DMZ	imaps	993	accept
intranet	Internet	http	80	accept
any	any	any	any	deny

Table I
A SIMPLE FIREWALL POLICY

Figure 1 illustrates a widely-used setup of a firewall, separating three networks: the external Internet, the internal network that has to be protected (intranet), and an intermediate network, the demilitarized zone (DMZ). The DMZ is usually used for servers (e. g. the Web server and the Mail server) that should be accessible both from the outside (Internet) and the from internal network (intranet) and thus are governed by a more relaxed policy than the intranet. Table I shows a firewall policy as it can be found in security textbooks. Such a policy description uses a first-fit matching strategy, i. e. the first match overrides later ones. For example, a packet from the Internet to the intranet is rejected (it only matches the last line of the table), whereas an smtp-packet from the intranet to the DMZ is accepted (third line of Table I). Lines in such a table are also called *rules*; together, they describe the *policy* of a firewall. We will use the terms policy and rules synonymously.

1) Packets and Networks: As a prerequisite, we need a formal model of packets and networks. As we do not want to depend on a specific representation of addresses, we introduce the abstract types $(\alpha :: \text{adr})\ \text{src}$ and $(\alpha :: \text{adr})\ \text{dest}$ for the source and destination address. Here, $\alpha :: \text{adr}$ restricts the types that can be used to instantiate α to members of the (unconstrained) type class adr. Further, we introduce $\beta\,\text{content}$ for the content (payload) of a packet and id for the a unique identifier of a packet. The type of a packet can be defined as:

$$\textbf{types}\ \ (\alpha,\beta)\,\text{packet} = \text{id}\ \times (\alpha{::}\text{adr})\ \text{src} \\ \times\ (\alpha{::}\text{adr})\ \text{dest}\ \times \beta\,\text{content} \tag{1}$$

Further, we define projectors, e. g. getId, for accessing the components of a packet. We model networks, or just *nets*, as sets of sets of addresses, i. e.

$$\textbf{types}\qquad \alpha\ \text{net} = (\alpha{::}\text{adr})\,\text{set}\,\text{set} \tag{2}$$

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

$$\begin{array}{lll} \textbf{definition} & \_ \sqsubset \_ & :: (\alpha{::}\text{adr}) \Rightarrow \alpha\,\text{net} \Rightarrow \text{bool} \\ \quad\textbf{where} & a \sqsubset S & \equiv \exists s \in S.\ (a \in s) \end{array} \tag{3}$$

2) Address Representations: So far, we have left the concrete address format generic; our model allows different representations. Some of them were presented in [3]. Here, we concretize one possible address format, namely IPv4 addresses together with ports. In this setting, an address consists of a 32 bit number, represented as four-tuple of Integers, and a port.

$$\begin{array}{lll} \textbf{types} & \text{ip} & = \text{int} \times \text{int} \times \text{int} \times \text{int} \\ & \text{port} & = \text{int} \\ & \text{ipv4} & = \text{ip} \times \text{port} \end{array} \tag{4}$$

As we only model the transport layer, i. e., TCP/IP, we do not model application level protocols, e. g. http, explicitly. Overall, application level protocols are, on this level, only visible by the destination port, e. g., http packets usually have a destination port 80.

3) The Firewall Policy: From an abstract point of view, a policy is a partial function of packets to decisions, e. g. deny or accept labelled with translated input data. The datatype:

$$\textbf{datatype}\qquad \alpha\,\text{out} = \text{accept}\,\alpha \mid \text{deny}\,\alpha \tag{5}$$

for decisions allows for modeling the modifications of return packets; thus, our model can capture address-translation techniques (such as network address translation (NAT)), realized by some firewalls, as well by providing suitable combinators. The type of a policy follows directly from this:

$$\begin{array}{l} \textbf{types} \\ \quad (\alpha,\beta)\,\text{policy} = (\alpha,\beta)\,\text{packet} \rightharpoonup ((\alpha,\beta)\,\text{packet})\,\text{out} \end{array} \tag{6}$$

where $\alpha \rightharpoonup \beta$ denotes partial functions (recall Section II-A).

Moreover, the override operator for partial functions $(\_ +\!\!+ \_)$ allows several rules to be combined to form a policy. For example, $r_2 +\!\!+ r_1$ combines the rules r_1 and r_2, where r_1 overrides (has higher precedence than) r_2. We can then define several *rule combinators* that substantially simplify the formalization of a concrete policy. For example, the usual "catch-all" rule for denying all traffic is expressed as:

$$\begin{array}{lll} \textbf{definition} & \text{deny\_all} & :: (\alpha,\beta)\,\text{policy} \\ \quad\textbf{where} & \text{deny\_all}\,p & \equiv \text{Some}(\text{deny}\,p) \end{array} \tag{7}$$

Many other combinators for restricting traffic based on a packet's source and destination can already be defined on this abstraction level. A rule allowing all packets coming from network s can be defined as

$$\begin{array}{l} \textbf{definition} \\ \quad \text{allow\_all\_from} :: \alpha\,\text{net} \Rightarrow (\alpha,\beta)\,\text{policy} \\ \textbf{where} \\ \quad \text{allow\_all\_from}\,s \equiv \text{Some}(\text{allow\_all} \restriction_{\{p \mid (\text{getSrc}\,p) \sqsubset s\}}) \end{array} \tag{8}$$

where $\_ \restriction \_$ is the restriction operator on partial functions.

B. Modeling a Policy

Our abstract firewall model presented in the last section, allows for the direct formalization of the informal policy given in Table I. First we have to define the subnets of type ipv4 net based on their IP address ranges, e. g.:

$$\text{intranet} \equiv \left\{ \left\{ ((a,b,c,d),p) \middle| \; (a=172) \wedge (b=168) \right\} \right\} \text{ and}$$

$$\text{dmz} \equiv \left\{ \left\{ ((a,b,c,d),p) \middle| \; \begin{array}{c}(a=172)\\ \wedge(b=16)\wedge(c=70)\end{array} \right\} \right\}.$$

We can then define the rules of our policy. Using the provided combinators, this can be done in a similar way to that used with many firewall configuration tools. The only slight difference is that in our scenario, the rules are treated backwards. The policy shown in Table I is represented in our combinator language as follows:

definition
$$\begin{aligned}\text{Policy} \equiv{}& \text{deny\_all}\\ &\mathbin{++} \text{allow\_port\_from\_to intranet internet } 80\\ &\mathbin{++} \text{allow\_port\_from\_to intranet dmz } 993\\ &\mathbin{++} \text{allow\_port\_from\_to dmz intranet } 25\\ &\mathbin{++} \text{allow\_port\_from\_to intranet dmz } 25\\ &\mathbin{++} \text{allow\_port\_from\_to internet dmz } 80\\ &\mathbin{++} \text{allow\_port\_from\_to internet dmz } 25\end{aligned}$$

C. Testing Stateless Firewalls: Direct Approach

The *test specification* for the stateless firewall case is now within reach: we just state that the *firewall under test (fut)* has the same filtering function behavior as our policy:

$$\text{testspec test:} \qquad \mathit{fut}\ x = \text{Policy}\ x \tag{9}$$

Usually, this test specification will be 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 content. Another predicate can ensure that only packets which cross network boundaries are considered.

Applying our test case generation (recall Section II-B), we get 72 test cases. Among them are the following two, where the first is a packet meant to be rejected by the firewall, while the second one should be passed unchanged:

1) $\mathit{fut}(12,((7,13,12,10),6),((172,168,2,1),80),\text{content}) = \text{Some}(\text{deny}(12,((7,13,12,10),6),((172,168,2,1),80)\ \text{content}))$
2) $\mathit{fut}(8,((172,168,12,13),12),((172,16,70,10),25),\text{content}) = \text{Some}(\text{accept}(8,((172,168,12,13),12),((172,16,70,10),25),\text{content}))$

The Some is a consequence of our representation of policies as partial functions, and means here that the mapping is defined for this packet. The test data could then be fed into a real firewall test driver. Overall, testing stateless packet filters is quite similar to classical unit testing of stateless software.

IV. Transforming Policies

Policy transformations are functions that map policies to policies. We decided to represent policy transformations as *syntactic rules*; this choice paves the way for expressing the entire normalization process inside HOL by functions manipulating abstract policy syntax. For the sake of conciseness, however, we will introduce only a few core combinators.

datatype
$$\begin{aligned}(\alpha,\beta)\,\text{Comb} = {}& \text{DenyAll} \mid \text{DenyFromTo}\ \alpha\ \alpha\\ &\mid \text{AllowPort}\ \alpha\ \alpha\ \beta\\ &\mid \text{Conc}((\alpha,\beta)\,\text{Comb})\ ((\alpha,\beta)\,\text{Comb})\end{aligned} \tag{10}$$

Moreover, we introduce the infix notation, $\_ \oplus \_$, for the constructor Conc _ _. Next, we define a semantic interpretation, C, from "syntactic" combinators to the "semantic" ones presented in the previous section:

fun $C :: (\text{ipv4 net}, \text{port})\ \text{Comb} \Rightarrow (\text{ipv4}, \text{Content})\ \text{Policy}$
where
$$\begin{aligned}&C\ \text{DenyAll} &&= \text{deny\_all}\\ &C\ (\text{DenyFromTo}\ x\ y) &&= \text{deny\_all\_from\_to}\ x\ y\\ &C\ (\text{AllowPort}\ x\ y\ p) &&= \text{allow\_port\_from\_to}\ x\ y\ p\\ &C\ (x \oplus y) &&= C\ x \mathbin{++} C\ y\end{aligned} \tag{11}$$

For the sake of simplicity, we fix here the address and content representation of a packet. However, the transformation works equally well for other representations. From these definitions follows a natural way to define and prove the correctness of a transformation. For example, the semantics of the following two policies is equivalent:

lemma redundant_allowPort:
$$\begin{aligned}&C\ (\text{AllowPort}\ x\ y\ p \oplus \text{DenyFromTo}\ x\ y)\\ &= C\ (\text{DenyFromTo}\ x\ y)\end{aligned} \tag{12}$$

The proof essentially involves applying the semantic interpretation function and unfolding the definitions of the semantic operators.

A. Elementary Transformation Rules

A large collection of elementary policy transformation rules can be proved correct in a similar way. For example, we can show that a shadowed rule is redundant (Equation 13), the $\_\oplus\_$ operator is associative (Equation 14), and, in many cases, commutativity holds (Equation 15-Equation 17):

$$C\ (a \oplus \text{DenyAll}) = C\ \text{DenyAll} \tag{13}$$

$$C((a \oplus b) \oplus c) = C(a \oplus (b \oplus c)) \tag{14}$$

$$\begin{aligned}&C(\text{AllowPort}\ x\ y\ a \oplus \text{AllowPort}\ x\ y\ b)\\ &\quad = C(\text{AllowPort}\ x\ y\ b \oplus \text{AllowPort}\ x\ y\ a)\end{aligned} \tag{15}$$

$$\begin{aligned}&C(\text{DenyFromTo}\ x\ y \oplus \text{DenyFromTo}\ u\ v)\\ &\quad = C(\text{DenyFromTo}\ u\ v \oplus \text{DenyFromTo}\ x\ y)\end{aligned} \tag{16}$$

$$\text{dom}(C\ a) \cap \text{dom}(C\ b) = \{\} \Longrightarrow C(a \oplus b) = C(b \oplus a) \tag{17}$$

B. Complex Transformation Rules

Elementary rules can be organized to compute entire normal forms on policies. Instead of implementing these computations inside the prover (i. e. by a tactic program), we can define these functions entirely inside HOL, which gives us the possibility to prove their termination and completeness formally. In fact, we need nine such complex transformation rules for our normalization (see Section V). In our paper we will present just one in more detail, namely the sorting phase transformation; the interested reader is referred to the HOL-TESTGEN project page to see the complete proofs.

The sorting phase is formalized as a function mapping combinator lists to combinator lists by sorting them according to the following principles:

- If there is a DenyAll, it should be at the first place.
- Rules dealing with the same set of networks should be grouped together, with the DenyFromTo's coming before the AllowPort's.

The semantic correctness of this transformation holds only under certain conditions that have to be established by previous phases of the normalization.

The sorting function itself is a slightly adapted version of a standard function from the Isabelle library. The ordering relation however, has to be defined appropriately according to our problem domain:

$$\begin{aligned}
&\textsf{fun}\ \ \mathrm{insort}\ \ \textsf{where}\\
&\quad \mathrm{insort}\ \ a\ l\ [] &&= [a]\\
&\quad \mathrm{insort}\ \ a\ l\ (x\#xs) &&= \mathrm{if}\ \ \mathrm{smaller}\ \ a\ x\ l\\
&&&\quad\ \mathrm{then}\ \ a\#x\#xs\\
&&&\quad\ \mathrm{else}\ \ x\#(\mathrm{insort}\ \ a\ l\ xs)
\end{aligned} \tag{18}$$

$$\begin{aligned}
&\textsf{fun}\ \ \mathrm{sort}\ \ \textsf{where}\\
&\quad \mathrm{sort}\ \ l\ [] &&= []\\
&\quad \mathrm{sort}\ \ l\ (x\#xs) &&= \mathrm{insort}\ \ x\ l\ (\mathrm{sort}\ \ l\ xs)
\end{aligned} \tag{19}$$

$$\begin{aligned}
&\textsf{fun}(\mathrm{sequential})\quad \mathrm{smaller}\ \ \textsf{where}\\
&\ \ \mathrm{smaller}\ \ \mathrm{DenyAll}\ \ x\ l\ = \mathrm{true}\\
&\ \ \mathrm{smaller}\ \ x\ \ \mathrm{DenyAll}\ \ l\ = \mathrm{false}\\
&\ \ \mathrm{smaller}\ \ x\ y\ l = (x = y) \vee\\
&\quad \mathrm{if}\ \ \mathrm{bothNet}\ \ x = \mathrm{bothNet}\ \ y\\
&\quad \mathrm{then}\ \ \mathrm{case}\, y\ \ \mathrm{of}\ \ \mathrm{DenyFromTo}\ \ a\ b\\
&\qquad\quad \Rightarrow x = \mathrm{DenyFromTo}\ \ b\ a \mid \_ \Rightarrow \mathrm{true}\\
&\quad \mathrm{else}\ \ \mathrm{pos}\ (\mathrm{bothNet}\ \ x)\ l \leq \mathrm{pos}(\mathrm{bothNet}\ \ y)\ l)
\end{aligned} \tag{20}$$

Here, 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 is a list of network sets. The ordering of this list determines which group of rules is treated as smaller than another. According to the requirements given to the smaller relation, the concrete ordering of this list is irrelevant. However, we will usually need the requirement that all the sets of network pairs of a policy are in this list. This requirement is formalized in a predicate called all_in_list.

First, we have to provide a proof that after applying sort using this smaller relation, we will get a sorted list. The main prerequisites for this proof are that the relation is transitive, reflexive, and antisymmetric. This is true as long as all the network pairs that appear in the policy are in the list l. Next, we have to prove that this transformation is semantics-preserving. We make the following assumptions about the input policy:

- singleComb: The policy is given as a list of single rules.
- allNetsDistinct: All the networks are either equal or disjoint.
- wellformed_policy1: There is exactly one DenyAll and it is at the first position.
- wellformed_policy3: The domain of an AllowPort rule is disjoint from all the rules appearing on the right of it.

In the next section, we show that these conditions do always hold at that specific point in our normalization algorithm. To facilitate the proofs, we further define a matching function that returns Some r for the first rule r in a policy p that matches packet x and None otherwise.

$$\begin{aligned}
&\textsf{fun}\ \ \mathrm{mr\_rev}\ \ \textsf{where}\\
&\quad \mathrm{mr\_rev}\ \ a\ (x\#xs) &&= \mathrm{if}\ \ a \in \mathrm{dom}(\mathrm{C}\ \ x)\\
&&&\quad\ \mathrm{then}\ (\mathrm{Some}\ \ x)\\
&&&\quad\ \mathrm{else}\ (\mathrm{mr\_rev}\ \ a\ xs)\\
&\quad \mathrm{mr\_rev}\ \ a\ [\,] &&= \mathrm{None}
\end{aligned} \tag{21}$$

$$\begin{aligned}
&\textsf{definition}\quad \mathrm{matching\_rule}\ \textsf{where}\\
&\qquad \mathrm{matching\_rule}\ \ a\ x \equiv \mathrm{mr\_rev}\ \ a\ (\mathrm{rev}\, x)
\end{aligned} \tag{22}$$

To prove our main theorem, the semantic correctness of sort, we apply an indirect approach. We show that for all packets x, and for all policies p and s that satisfy the conditions listed above, 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. If we can additionally prove that the sorting function preserves these conditions, we can establish the main theorem:

```
lemma C_eq_Sets_mr:
  assumes sets_eq:  set p = set s
  and wp1_p:  wellformed_policy1  p
  and wp1_s:  wellformed_policy1  s
  and wp3_p:  wellformed_policy3  p
  and wp3_s:  wellformed_policy3  s
  and aND:  allNetsDistinct  p
  and SC:  singleComb  p
shows  matching_rule  x p = matching_rule  x s
```

a) Proof Sketch.: A case distinction over matching_rule x p results in two cases: this expression can either be None or Some y:

1) None. This case is shown by contradiction. As there is a DenyAll which matches all packets, the matching rule of x cannot be None.
2) Some y. This is shown by case distinction on y, i. e. on the matching rule. By our definition of the Comb datatype, there are four possibilities:
   - DenyAll: If the matching rule is DenyAll, there is no other rule in p which matches, as it is necessarily the last considered rule. Thus, as the two sets are equal, there is no other rule in s that matches either. As

DenyAll is in s, this has to be the matching rule in s.

- 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 DenyAll is the last considered rule, the only possibility is an AllowPort a b c, for some c. However, due to wellformed_policy3, such a rule can only be on the right of the matching rule as their domains are not disjoint. But if it were on the right, the allow-rule would be the matching rule. Thus there is no such rule. Since such a rule does not exist in p, it does not exist in s either. As DenyFromTo a b is in s, this has to be the matching rule in s.
- AllowPort a b c: This sub-proof is very similar: The only alternative rule which could match is the corresponding DenyFromTo a b, but if this rule appears on the right of the matching rule, this would contradict the wellformedness conditions. Therefore, this has to be the matching rule also in s.
- $a \oplus b$: This case is ruled out by the wellformedness conditions.

The formal proof of this lemma can be found in the appendix.

Our main correctness theorem of the sorting phase is presented as follows:

lemma C_eq_sorted: assumes ail: all_in_list p l
  and wp1: wellformed_policy1 p
  and wp3: wellformed_policy3 p
  and aND: allNetsDistinct p
  and SC: singleComb p
  shows C (list2policy(sort l p)) = C(list2policy p)

We thus have a sorting algorithm for policies that is proven to be semantics-preserving given that some well-specified conditions hold for the input policy.

V. Normalizing Policies

The sorting phase is only one part of the full normalization, which is organized into nine phases in total. We impose the following two restrictions on the input policies:

- Each policy must contain a DenyAll rule. If this restriction were to be lifted, the 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 were to be lifted, we would need some additional phases before the start of the normalization procedure presented below. This rule would split single rules into several 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.

As a result, the procedure generates a list of policies, in which:

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

This result is desirable since the test case generation for rules between networks A and B is independent of the rules that specify the behavior 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 to minimize 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 full procedure can be written as follows:

definition

$$\begin{aligned}\text{normalize}\, p \equiv (&\text{removeAllDuplicates} \circ \text{insertDenies}\\ &\circ \text{separate} \circ (\text{sort}\ (\text{Nets\_List}\ p))\\ &\circ \text{removeShadowRules2}\\ &\circ \text{remdups} \circ \text{removeShadowRules3}\\ &\circ \text{removeShadowRules1} \circ \text{policy2list})\ p\end{aligned} \quad (23)$$

The normalization procedure consists of nine transformation functions that are processed in sequence. In the following, we discuss the nine phases and their preconditions in more detail. These preconditions are necessary to establish the correctness theorem for the overall procedure:

theorem C_eq_normalize:
  assumes a1: member DenyAll p
    and a2: allNetsDistinct p
shows C(list2policy(normalize p)) = C p

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

1) **policy2list**: transforms a policy into a list of single rules. The result does not contain policies composed via _ ⊕ _, i. e. the property singleComb is established.
2) **removeShadowRules1**: removes all the rules that appear in front of a DenyAll. The transformation preserves singleComb and establishes wellformed_policy1, stating that there is a DenyAll in front 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 last one of the list remains. As policies are evaluated from right to left, 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 wellformed_policy3, the invariant necessary for the sorting function to be correct. It ensures that the domain of an AllowPort rule is disjoint from all the rules appearing on the right of it.
6) **sort**: maintains (see previous section) 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. As already mentioned, the order of the grouped rules is considered to be irrelevant to correctness. However, a user-defined ordering might give room for further optimizations. This can be done very easily by stating manually a list with sets of network pairs. All properties and side-conditions are still valid as long as we can prove that this list contains all the sets of network pairs appearing in the policy.

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 singleComb is invalidated by this transformation, all the others are maintained. We do get two important additional properties:
   - OnlyTwoNets: Each policy treats at most two different networks.
   - separated: The domains of the policies are pairwise disjoint (except from DenyAll). This follows from the fact that the rules were already grouped.
8) **insertDenies**: is the only phase where additional rules are inserted. In front of each policy, we add the two rules denying all traffic between those two networks in both directions. As we add new rules here, the proof of semantic equivalence is relatively involved. The main part consists in proving that separated is maintained with this transformation. This phase is only semantics-preserving because of the initial requirement that the policy has a DenyAll. Only after this phase, the traffic between two networks is characterized completely by the corresponding policy.
9) **removeAllDuplicates**: removes superfluous duplicate DenyFromTo's introduced by the previous phase.

After the last phase, we get a list of policies that satisfies the requirements stated in the beginning of this section. Using the correctness of the phases, it is straightforward to prove semantic equivalence of the full procedure. 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 the test cases of all the smaller policies.

A. An Example

In the following we show the effect of the normalization procedure on our small example policy. The original policy contains 7 rules restricting the traffic between three networks:

definition

$$\begin{aligned}\text{Policy} \equiv\ & \text{DenyAll}\\ & \oplus \text{AllowPort intranet internet } 80\\ & \oplus \text{AllowPort intranet dmz } 993\\ & \oplus \text{AllowPort dmz intranet } 25\\ & \oplus \text{AllowPort intranet dmz } 25\\ & \oplus \text{AllowPort internet dmz } 80\\ & \oplus \text{AllowPort internet dmz } 25\end{aligned}$$

After applying the normalization procedure, we get:

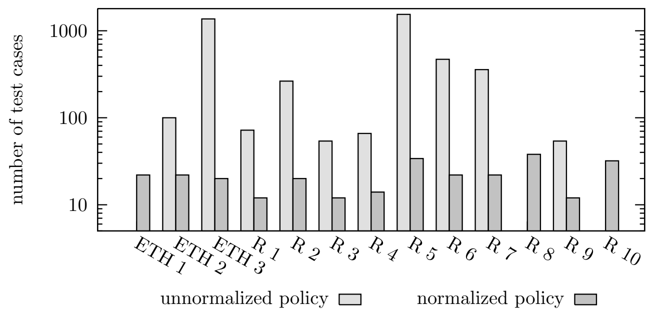

Figure 2. The normalization of policies decreases the number of test cases by several orders of magnitude.

1) DenyAll
2) DenyFromTo intranet internet
   $\oplus$ DenyFromTo internet intranet
   $\oplus$ AllowPort intranet internet 80
3) DenyFromTo intranet dmz
   $\oplus$ DenyFromTo dmz intranet
   $\oplus$ AllowPort intranet dmz 993
   $\oplus$ AllowPort dmz intranet 25
   $\oplus$ AllowPort intranet dmz 25
4) DenyFromTo internet dmz
   $\oplus$ DenyFromTo dmz internet
   $\oplus$ AllowPort internet dmz 80
   $\oplus$ AllowPort internet dmz 25

Now, there are four policies with a total of 13 rules. The number of generated test cases shrinks from 92 to 17, the time required to generate them from 40 to about 2 seconds. It is important to note that in general we do not generate a policy for those partitions which do not have a rule in the initial policy. They are treated only by the catch-all rule. This might easily be changed if that turned out to be useful - either by adjusting the insertDenies phase or by adding a new phase right after that one.

VI. Empirical Results

In this section, we report on several case studies that show the benefit of policy normalization with respect to both the number of test cases generated and the runtime required for generating those test cases.

In a first experiment, we explored the impact of policy normalization on the overall time required for generating test cases (see Table II) on both policies that are actually used at ETH Zürich and randomly generated ones. The most important result is that while the overall number of rules can increase during normalization, the overall time required for generating test cases is significantly smaller after normalization, even if we include the time needed for normalization. While for some policies, the test case generation took more than 24 hours, the required time for normalizing the policy and generating test cases for the normalized policy is only a few seconds. The overall 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

		ETH 1	ETH 2	ETH 3	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Not Normalized	Networks	4	6	3	2	3	4	4	4	3	5	5	6	6
	Rules	11	9	12	13	9	5	7	13	13	8	15	5	10
	TC Generation Time (sec)	>24h	22	26382	10	187	6	9	59364	1388	646	>24h	8	>24h
	Test Cases	—	100	1368	72	264	54	66	1544	470	358	—	54	—
Normalized	Rules	17	16	14	8	14	11	10	24	26	17	28	11	25
	Segments	6	5	4	2	4	5	3	7	4	6	9	5	9
	Normalization (sec)	0.5	0.3	0.6	0.5	0.4	0.2	0.3	1.1	0.8	0.3	1.4	0.2	0.8
	TC Generation Time (sec)	0.8	0.7	0.9	0.5	0.6	0.3	0.4	1.2	0.7	0.7	1.4	0.4	0.8
	Test Cases	22	22	20	12	20	12	14	34	22	22	38	14	32

Table II
TEST CASE GENERATION FOR FIREWALL POLICIES UTILIZING NORMALIZATION.

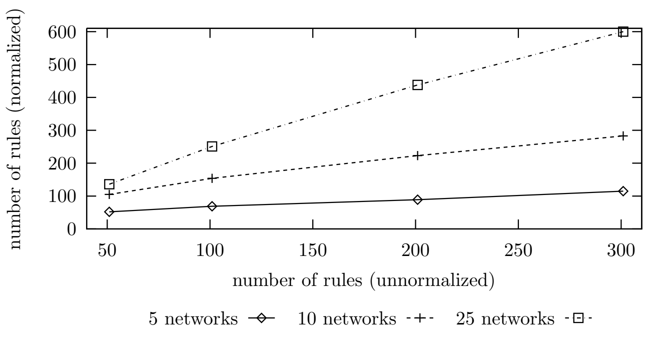

Figure 3. The size (number of rules) of a policy after normalization increases with both the number of rules in the unnormalized policy and the number of networks.

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 (see Figure 2), 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, we investigated the effect of normalization in isolation. Table III 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 3). All policies start with a DenyAll rule and one in every five rules is a DenyFromTo. 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 DenyFromTo's to be inserted. In the opposite case, it decreases a lot, as many of those 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.

We also generated test cases for one segment (i. e. a normalized policy between exactly two networks) in isolation: the test case generation for a policy with 50 rules only takes about 3 minutes and more than 100 rules can still be processed in less than an hour. Thus, test case generation for very large policies seems to be feasible, even though policy normalization can increase the overall number of rules.

VII. CONCLUSION AND RELATED WORK

A. Related Work

Several approaches for specification-based testing of firewalls have been proposed. El-Atawy et al. [9, 10] present a policy segmentation technique. They also give some measurements to the segments such that important segments can be tested more rigorously. Jürjens and Wimmel [11] propose a specification-based testing of firewalls that employs a formal model of the network and automatically derives test cases. 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. [12] 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 [13] propose a policy-based host classification which can be used to detect errors and anomalies in a firewall policy. A common feature of all these approaches is, that they do not address the transformation of the specification, i. e. the policy.

In contrast, transforming security policies, e. g. ones based on XACML [14], to improve their overall runtime performance is a well-known technique [15, 16]. This approach also been extended to other policy languages. In particular, Liu et al. [16] 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. [17] 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 is that of Harman et al. [18], which is further developed in, e. g. [19–21]. They introduce the concept of *testability transformations*, i. e. source-to-source transformations of programs that increase their testability. Similar to our work, 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 transformation of the source code, i. e. the system under test, we transform a formal specification, which makes

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
Partitions	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]	18	63	37	50	367	626	155	1897	8260	404	3184	44415

Table III
THE SIZE OF A NORMALIZED POLICY MAINLY DEPENDS ON THE NUMBER OF RULES IN RELATION TO THE NUMBER OF NETWORKS.

our approach applicable in black-box testing scenarios.

Moreover, we provide a uniform, tool-supported approach for formally analyzing and transforming a test-specification, and generating test cases for an implementation. And last but not least, our approach is based on a formal proof that the applied testability transformation preserves semantics.

B. Conclusion and Future Work

Harman [20] discusses a list of open problems in testability transformations. With our work, we make a significant contribution to the final problem in his list: testability transformations for specification-based testing. In particular, we have presented a testability translation for model-based generation of test cases for firewalls. Moreover, we proved formally the correctness, i. e. semantics preservation, of transformations within the same framework used for test case generation. In HOL-TESTGEN/FW, we believe we have developed the first domain-specific test tool to integrate the specification, formal analysis, transformation, and test case generation of firewall policies.

Besides applying test case generation to other kind of security domains, e. g. [22], we see several productive lines of future work. In [8], we used sequence testing techniques for testing stateful firewalls using HOL-TESTGEN. 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 [20]: 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. [21], requires the development of new test hypotheses. Here, the integrated approach of HOL-TESTGEN allows for the formal computation and analysis of such hypotheses. Finally, the relation between testability transformation and fault models needs to be investigated further.

On the practical side, several extensions are envisaged: first, our integrated test-harness generator could be configured for generating test data in a format that is suitable for packet injection tools. This would allow for testing the conformance of deployed firewalls. Second, following model-driven approaches, e. g. [23], our policy specification can be used for generating configurations of different firewall implementations. This would allow the use of HOL-TESTGEN/FW for testing, optimization (both with respect of testability and performance), and configuration of real firewalls. And thirdly, integration of HOL-TESTGEN/FW with standard firewall configuration tools would increase their appeal to end-users. Like the second approach, this would allow a policy specification language to be used for test and configuration of a real firewall.

REFERENCES

[1] "Securing cyberspace for the 44th presidency," Center for Strategic and International Studies (CSIS), Tech. Rep., 2008.

[2] "Misconfigured router causes increased BGP traffic and isolated outages for internet services," http://tools.cisco.com/security/center/viewAlert.x?alertId=17657, 2009.

[3] A. D. Brucker, L. Brügger, and B. Wolff, "Model-based firewall conformance testing," in *Testcom/FATES 2008*, ser. LNCS, K. Suzuki and T. Higashino, Eds. Springer-Verlag, 2008, no. 5047, pp. 103–118.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer-Verlag, 2002, vol. 2283.

[5] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940.

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

[7] A. D. Brucker and B. Wolff, "Symbolic test case generation for primitive recursive functions," in *Formal Approaches to Testing of Software*, ser. LNCS, J. Grabowski and B. Nielsen, Eds. Springer-Verlag, 2004, no. 3395, pp. 16–32.

[8] ——, "Test-sequence generation with HOL-TESTGEN – with an application to firewall testing," in *TAP 2007: Tests And Proofs*, ser. LNCS, B. Meyer and Y. Gurevich, Eds. Springer-Verlag, 2007, no. 4454, pp. 149–168.

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

[10] 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*. IEEE Computer Society, 2007, pp. 151–160.

[11] J. Jürjens and G. Wimmel, "Specification-based testing of firewalls," in *Ershov Memorial Conference*, ser. LNCS, D. Bjørner, M. Broy, and A. V. Zamulin, Eds., vol. 2244. Springer-Verlag, 2001, pp. 308–316.

[12] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with logic: HOL spec-

ification and symbolic-evaluation testing for TCP implementations," in *POPL*, J. G. Morrisett and S. L. P. Jones, Eds. ACM Press, 2006, pp. 55–66.

[13] R. Marmorstein and P. Kearns, "Firewall analysis with policy-based host classification," in LISA. USENIX Association, 2006, pp. 4–4.

[14] "eXtensible Access Control Markup Language (XACML), version 2.0," 2005. [Online]. Available: http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-NORMATIVE.zip

[15] P. Miseldine, "Automated XACML policy reconfiguration for evaluation optimisation," in *Software Engineering for Secure Systems (SESS)*, B. D. Win, S.-W. Lee, and M. Monga, Eds. ACM Press, 2008, pp. 1–8.

[16] A. X. Liu, E. Torng, and C. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *IEEE Conference on Computer Communications (Infocom)*, 2008.

[17] R. Dssouli, K. Karoui, K. Saleh, and O. Cherkaoui, "Communications software design for testability: specification transformations and testability measures," *Information and Software Technology*, vol. 41, no. 11-12, pp. 729–743, 1999.

[18] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 3–16, 2004.

[19] R. M. Hierons, M. Harman, and C. Fox, "Branch-coverage testability transformation for unstructured programs," *Comput. J.*, vol. 48, no. 4, pp. 421–436, 2005.

[20] M. Harman, "Open problems in testability transformation," in *Software Testing Verification and Validation Workshop (ICSTW)*, 2008, pp. 196–209.

[21] 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 *Formal Methods and Testing*, ser. LNCS, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949. Springer-Verlag, 2008, pp. 320–344.

[22] A. D. Brucker and B. Wolff, "A verification approach for applied system security," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 3, pp. 233–247, 2005.

[23] A. D. Brucker, J. Doser, and B. Wolff, "An MDA framework supporting OCL," *Electronic Communications of the EASST*, vol. 5, 2006.

Appendix

```
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_notinSet foo2 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
```