Analysis of application-layer filtering policies with application to HTTP

(Article begins on next page)

Analysis of Application-Layer Filtering Policies with application to HTTP

Cataldo Basile, Antonio Lioy

***Abstract*—Application firewalls are increasingly used to inspect upper layer protocols (as HTTP) that are target or vehicle of several attacks and are not properly addressed by network firewalls. Like other security controls, application firewalls need to be carefully configured, as errors have a significant impact on service security and availability. However, currently no technique is available to analyse their configuration for correctness and consistency. This paper extends a previous model for analysis of packet filters to the policy anomaly analysis in application firewalls. Both rule pair and multi-rule anomalies are detected, hence reducing the likelihood of conflicting and suboptimal configurations. The expressiveness of this model has been successfully tested against the features of Squid, a popular web caching proxy offering various access control capabilities. The tool implementing this model has been tested on various scenarios and exhibits good performance.**

***Index Terms*—firewall, application gateway, proxy, policy anomalies, policy conflicts, regular expressions.**

I. Introduction

Despite huge investments in security, US government agencies report that billions of dollars are lost every year due to cyber attacks [1], and those based on malware and web vulnerabilities are amongst the most damaging ones [2].

Firewalls are traditionally an important component of a cyber defense architecture and frequently take the form of packet filters. However, as threats evolve and increasingly target the higher OSI levels, users turn to a new firewall class: the application gateway. This analyzes application-level protocols and payloads, and hence can enforce sophisticated policies and thwart attacks that packet filters are unable to prevent. Application firewalls are often part of a reverse proxy (to shield a public server from attacks) or a forward proxy (to authenticate the internal clients for external access and filter their requests according to the company policy).

The presence of a firewall is no guarantee of protection unless it is properly configured. Writing a firewall policy is a security-sensitive and error-prone activity: as assessed by the NSA [3], "inappropriate or incorrect security configurations (most often caused by configuration errors at the local base level) were responsible for 80% of Air Force vulnerabilities".

While techniques and tools exist for the specification and analysis of packet filter configurations [4] [5], to the best of our knowledge no tool exists for application firewalls. We address this problem by extending our geometric model in [6] to a formal model of application-level policies that permits their anomaly analysis. This model is based on an IETF-compliant representation of the architecture of stateful and application firewalls [7] and it is inspired by the stateful model of Liu [8]. Our anomaly analysis permits the identification in application firewalls of *conflicting rules* (a symptom of a wrong configuration as they may be activated simultaneously but enforce different actions) and *unnecessary rules* (that can be removed as they do not affect the decisions but decrease the performance). While there is no published statistics about the frequency of these cases, our talks with various security managers point in one direction: firewall policies are monotonic increasing in size, as nobody is taking the risk to remove, compact, or rewrite existing rules created by a previous manager. If a policy works, then nobody will touch it. If a policy does not work, then new specific rules will be added to fix the issue but no old rule will be removed. Therefore the availability of an automatic analysis tool is important to detect problems, suggest appropriate changes and create confidence in their correctness. Even when no anomaly exists, the tool is valuable just to certify this case.

We verified the expressiveness of the our model against the access control features of Squid [9], a popular HTTP proxy that offers many filtering capabilities. This was used also to test the performance of the associated analysis tool: the experiments confirm that the model can be proficiently used in real scenarios. While we modelled and tested only HTTP firewalls, our model can be equally well applied to other text-based application protocols, as FTP and SMTP.

The paper is organized as follows: section II briefly introduces our geometric model; section III describes existing firewall categories and identifies information relevant for modelling the application firewall; section IV sketches the Squid filtering model and introduces Squid filtering rules to derive requirements for our model; section V presents our support for regular expressions and introduces the algorithms for anomaly analysis in policies with text-based rules; section VI presents a computational analysis of the algorithms; section VII presents our tool and its experimental results; section VIII discusses relevant work in the same area and, finally, section IX draws conclusions and provides hints for future work.

II. The geometric Model

According to RFC-3198 [10], a policy is "a set of rules to administer, manage, and control access to network resources". The IETF architecture for policy-based access control [7] uses two main architectural elements, the Policy Enforcement Point (PEP) and the Policy Decision Point (PDP). The latter is the logical entity making decisions about access request, while the former actually enforces these decisions. Decisions are based

C. Basile and A. Lioy are with the Politecnico di Torino, Dip. Automatica e Informatica, Torino (Italy); e-mail: {cataldo.basile, antonio.lioy}@polito.it

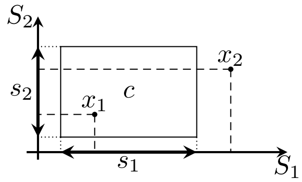


Fig. 1: Condition clause $c = s_1 \times s_2$ in a selection space formed by two selectors S_1 and S_2

on a policy (usually expressed as a ruleset) defining the decision criteria. We name *policy-enabled element* a component able to enforce a policy, that is, a component that acts as a PEP and knows how to contact its PDP.

Firewalls are common policy-enabled elements and have different capabilities to analyse network traffic and pass or block specific packets or flows. Packet filtering is the simplest control that a firewall may provide: decisions are made on each received packet based on information in its IP and transport headers, without looking at the global data stream.

We defined in [6] a geometric model of packet filter policies expressed as rulesets, useful for conflict analysis and policy translation. That model is summarized here as this paper extends it to higher network levels. Modelled policies are expressed as a set of rules in the "if *condition* then *action*" format [10]. Rules consist of a condition clause and an action clause. Actions are well known and organized in an *action set* $\mathcal{A}$. For filtering devices, the enforceable actions are Allow and Deny, thus $\mathcal{A} = \{A, D\}$.

Conditions are typed predicates concerning a given selector. A selector describes the values that a protocol field may take, e.g. the IP source selector is the set of all possible IP source addresses. Geometrically a condition is a subset of its selector for which it evaluates to true. A condition on a given selector matches a packet if the value of the field referred to by the selector belongs to the condition. For instance, in Fig. 1 the conditions are $s_1 \subseteq S_1$ and $s_2 \subseteq S_2$ (on the axes), both s_1 and s_2 match the packet x_1, while only s_2 matches x_2.

To consider conditions in different selectors, the decision space is extended using the Cartesian product because distinct selectors refer to different fields, possibly from different protocol headers. Given a policy-enabled element that allows the definition of conditions on the selectors $S_1, S_2, \dots, S_\mathrm{m}$ (where m is the number of available selectors) its *selection space* is

$$\mathfrak{S} = S_1 \times S_2 \times \cdots \times S_\mathrm{m}$$

Accordingly, the *condition clause* c is a subset of $\mathfrak{S}$:

$$c = s_1 \times s_2 \times \cdots \times s_\mathrm{m} \subseteq S_1 \times S_2 \times \cdots \times S_\mathrm{m} = \mathfrak{S}$$

$\mathfrak{S}$ represents the totality of the packets, but not all its subsets are valid condition clauses: only hyper-rectangles or union of hyper-rectangles (obtained as the Cartesian product of conditions) are valid. This is an intrinsic constraint of the policy languages as they specify rules by defining a condition for each selector. Fig. 1 graphically represents a condition clause in a two-dimensional selection space. In the rest of the paper we will generically name hyper-rectangle both single (compact) ones and union of hyper-rectangles.

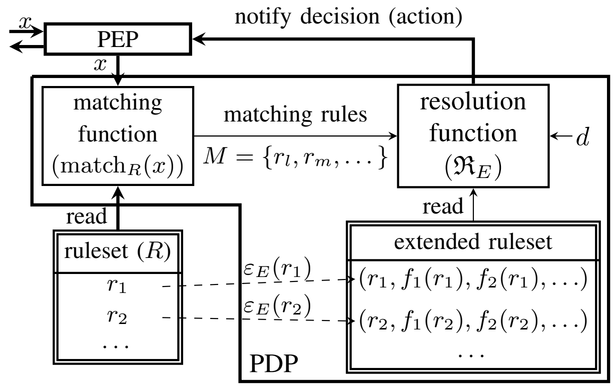


Fig. 2: Geometric policy model: the PDP.

In our model, a rule is expressed as $r = (c, a)$, where $c \subseteq \mathfrak{S}$ and $a \in \mathcal{A}$. A condition clause of a rule matches a packet, or briefly a rule matches a packet, if all the conditions forming the clause match the packet: in Fig. 1, the rule with condition clause c matches the packet x_1 but not x_2. Therefore, two or more rules match the same packet and can be activated simultaneously if the intersection of their condition clauses is non-empty. We will say that two rules intersect each other if their condition clauses do. This is useful for anomaly detection and policy transformation purposes (sections II-B and II-C).

The functional behaviour of the PDP is depicted in Fig. 2. When the PEP receives a packet x it throws an event and sends x to the PDP. The PDP identifies the set of rules $M = \{r_l, r_m, \dots\} \subseteq R$ that match x. This is formalized through the match_R function:

$$\begin{array}{rccl} \mathrm{match}_R : & \mathfrak{S} & \longrightarrow & 2^R \\ & x & \longmapsto & M = \{r_i \in R \mid x \in c_i\} \end{array}$$

that returns the subset $M \subseteq R$ of rules whose condition clauses match x. We'll use the form 2^R to denote the power set of R, the set of all subsets of R. The decision criteria for the action to apply when a packet matches two or more rules is abstracted by means of the *resolution strategy*

$$\mathfrak{R} : 2^R \to \mathcal{A}$$

Given a set of rules representing a policy, the resolution strategy maps all the possible groups of rules to an action $a \in \mathcal{A}$. When no rule matches a packet, the PDP selects the default action $d \in \mathcal{A}$. The decision is then notified to the PEP.

Resolution strategies may use, besides intrinsic rule data (i.e. condition clause and action clause), also "external data" related to each rule, such as priority, identity of the creator, and creation time. Formally, every rule r_i is extended through a function ε_E so that the rule becomes:

$$\varepsilon_E(r_i) = (r_i, f_1(r_i), f_2(r_i), \dots)$$

where $E = \{f_j : R \to X_j\}_j$ is a set of functions mapping rules to a set of external attributes X_j. In this case, the resolution strategy $\mathfrak{R}$ is the composition between the extension function ε_E, and a resolution function $\mathfrak{R}_E$ that works on the rule extensions, that is $\mathfrak{R} = \mathfrak{R}_E \circ \varepsilon_E$ (where $\circ$ is the function composition operation):

$$\mathfrak{R} : \quad \{r_l, r_m, \dots\} \quad \xmapsto{\varepsilon_E} \quad \{\varepsilon_E(r_l), \varepsilon_E(r_m), \dots\} \quad \xmapsto{\mathfrak{R}_E} \quad a$$

A policy is thus a function $p : \mathfrak{S} \to \mathcal{A}$ that connects each point of the selection space to an action taken from the action set $\mathcal{A}$ according to the rules in R. By defining $\mathfrak{R}(\emptyset) = d$ and $\mathfrak{R}(r_i) = a_i$, the policy p is formally defined as:

$$p(x) = \mathfrak{R}(\text{match}_R(x))$$

Therefore, a policy is completely defined by the 4-tuple $(R, \mathfrak{R}_E, E, d)$: the ruleset R, the resolution function $\mathfrak{R}_E$, the set E of mappings to the external attributes, and the default action d. Two policy representations $(R_1, \mathfrak{R}_{E_1}, E_1, d_1)$ and $(R_2, \mathfrak{R}_{E_2}, E_2, d_2)$ are *equivalent* if:

$$\forall x \in \mathfrak{S}, \mathfrak{R}_1(\text{match}_{R_1}(x)) = \mathfrak{R}_2(\text{match}_{R_2}(x))$$

where $\mathfrak{R}_1 = \mathfrak{R}_{E_1} \circ \varepsilon_{E_1}$ and $\mathfrak{R}_2 = \mathfrak{R}_{E_2} \circ \varepsilon_{E_2}$.

The use of this model in real cases requires a specific *model characterization*: all the enforceable actions, the needed selectors and the data type of each selector must be identified. For instance, Al-Shaer's five-tuple model can be easily represented introducing the action set $\mathcal{A} = \{A, D\}$, and five selectors: source and destination IP address (ips, ipd), source and destination port (ps, pd), and protocol type (proto). The IP addresses and the port numbers can be mapped to integers, and the protocol types to a set consisting of all the IANA registered protocols. Here is an example of a five-tuple rule:

```
( (ips = 1.2.3.4, ipd = 5.6.7.8, ps < 1024, pd = 80,
   proto = TCP), A)
```

We will use this syntax for rules in the rest of the paper.

A. Selector types

Current packet filters support three types of selectors: *exact match*, *range-based*, and *prefix match* ones [11].

Exact match selectors are unstructured sets with no specific order: elements can only be checked for equality. An example is the protocol type field of the IP header.

Range-based selectors are ordered sets where it is possible to naturally specify ranges as they can be easily mapped to integers. As an example, the ports in the TCP protocol are well represented using a range-based selector (e.g. 1024-65535).

Prefix match selectors are those without an explicit notion of ordering but such that ranges of values can be specified using a prefix regular expression. The typical case is the IP address selector (e.g. 10.10.1.*). As stated in [6], there is no need to distinguish between prefix match and range-based selectors as 10.10.1.* easily maps to [10.10.1.0, 10.10.1.255].

B. Anomalies

An abstract definition of policies presenting anomalies and algorithms to detect them is presented in [6]. Policies containing anomalies are divided in *conflicting policies*, such that for at least one point in the selection space two rules contradict each other, and *sub-optimal policies* that contain at least one rule that can be removed without changing the policy (*unnecessary rule*).

Conflicting policies are identified through *rule pair analysis*, that detects *correlated rules*: two rules r_i and r_j are correlated if $c_i \cap c_j \neq \emptyset$ and $a_i \neq a_j$. The effective contribution of a rule depends on all the other rules: a rule can be removed if its condition clause is completely covered by one rule or the union of several overriding rules. Therefore, an analysis limited only to rule pairs does not identify all the sub-optimal policies. The function $\text{eff}_p(r)$, which returns the portion of the rule r that "effectively" contributes to the policy p, is used to detect unnecessary rules:

$$\begin{array}{rrcl} \text{eff}_p: & R & \to & 2^{\mathfrak{S}} \\ & r & \mapsto & x \in c \text{ such that } \mathfrak{R}(\text{match}_R(x)) \neq \\ & & & \neq \mathfrak{R}(\text{match}_R(x) \setminus \{r\}) \end{array}$$

If $\text{eff}_p(r) = \emptyset$ then r is unnecessary for p, but two types of sub-optimality occur: the *general redundancy anomaly*, when the action of the unnecessary rules is always the same as the rules that cover it, and the *general shadowing anomaly*, when the policy enforces a different action for at least one point of the unnecessary rule.

C. Translating into low-level representations

The geometric model supports the definition of custom resolution strategies. However, existing policy-enabled elements typically use the FMR (First Matching Rule) resolution strategy. To this purpose, the *semantics-preserving policy morphism* has been introduced in [6]: given $(R, \mathfrak{R}_E, E, d)$, a morphism is a transformation that finds an equivalent policy $(R', \mathfrak{R}'_{E'}, E', d')$. Additionally, we demonstrated that every policy expressed in the geometric model can be translated into a policy that uses FMR.

III. Firewall types and model

As stated previously, the simplest feature of firewalls is packet filtering. *Stateful inspection* improves the packet filter functionality by also maintaining connection states at the transport layer, giving origin to *stateful firewalls*. Gouda and Liu presented in [8] a model of stateful firewalls as devices split into two components: the stateful section and the stateless one (Figure 3). The former analyses the transport headers and maintains a *state table* that associates a set of Boolean variables to each connection, usually represented by a five-tuple composed by the IP source and destination addresses, the source and destination port, and the L4 protocol. The state table is updated according to a set of stateful rules, depending on the received packet and the current table content. In the existing firewalls (e.g. iptables or commercial products), these rules are hardcoded and cannot be specified explicitly. From the functional point of view, this scenario can be modelled by

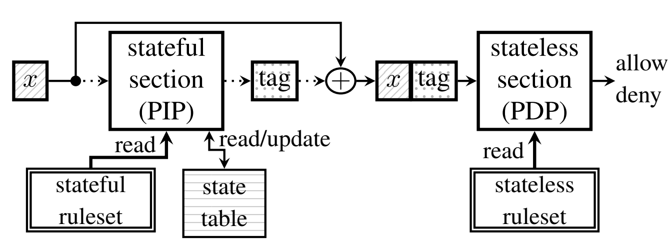


Fig. 3: Functional model of a stateful firewall (the $\oplus$ operation indicates concatenation).

associating each received packet to a set of Boolean variables, named *tag*, before handling control to the stateless section. Therefore, according to the AAA authorization framework in RFC-2904 [12], the stateful section does not take any decision, hence it does not perform any PDP functionality, rather it acts as a Policy Information Point (PIP) because it provides external information to the PDP.

The stateless section is the component that actually plays the PDP role in the firewall. It takes decisions according to a set of rules whose conditions apply to packet fields and also to tag values (for stateful firewalls only). By checking state information in the tag, the stateful firewalls can express more sophisticated rules, explicitly (e.g. allowing the traffic related to an "established" connection) and sometimes implicitly (e.g. blocking packets that violate the TCP specification).

Stateful firewalls may also enforce a simple form of bandwidth control: they may be used to specify the maximum number of connections allowed to a given destination, or the maximum packet rate per destination address or per port. Additionally, it is possible to bound the rule hits, i.e. the number of times that a rule is applied in a given time period. For example, this feature may be used to limit the number of ICMP packets allowed per second. We can model this case introducing another table that associates stateless rules to a set of counters. Some devices are also able to monitor the state of upper layer protocols, for instance, they can enforce a policy that only allows a DNS response if the corresponding query was seen first.

Another feature provided by some firewalls is *stateful protocol analysis*, also known as *deep packet inspection* (DPI), that is the ability to (recursively) extract nested information in TCP or UDP packets and take decisions using application protocol data and states. The firewalls with DPI capability are often referred to as *application firewalls*. When they have been tailored to a specific application, they are named *specialized application firewalls*. The most widespread specialized firewall is the Web Application Firewall (WAF), which deeply analyses HTTP traffic.

As in the case with stateful firewalls, DPI can also be modelled using the stateful and stateless sections. First, application firewalls are able to check the compliance to protocol standards or to a set of non-harmful implementations ("RFC compliance") and identify unexpected sequences of commands (e.g. repeated commands or those not preceded by other commands on which they depend). This means that they use protocol-specific stateful components. Application firewalls can also block the traffic depending on the values of protocol-specific properties and fields; for example, they can filter e-mail messages according to the attachment type, or block possibly harmful protocol operations, such as HTTP unsafe methods (e.g. TRACK, TRACE, DELETE). This enables fine-grained decisions, such as controlling access to a web page based not only on its URL but also on its content (e.g. checking for malicious Java or ActiveX applets), or blocking an SSL connection to a server presenting an untrustworthy certificate. Since information is often represented as character strings (e.g. MIME object, URL, filename), conditions in application firewalls are often formulated using regular expressions (e.g. Content-Type: image/*). In brief, the stateless section of application layer firewalls evaluates conditions on additional fields, which means that the decision space of stateful and application firewalls is composed of more selectors than in packet filtering and, in some cases, the evaluation of conditions is made using regular expressions.

Application data are typically split across several packets thus application firewalls are equipped with buffers to temporarily store and reassemble data flows. Consequently, not all the conditions can be evaluated every time a new packet arrives. Hence, the PEP will query the PDP not only when a packet is received but also when an entire *application protocol data unit* (APDU) is reconstructed. To avoid ambiguities, we will name protocol data unit (PDU) any data sent to the PDP to take decisions, e.g. an IP packet or an APDU.

An improved type of application firewall is the *application-proxy gateway*, which enforces an access control policy using a proxy agent. Application-proxy gateways keep track of authenticated users and can limit the maximum number of simultaneous users or connections per user.

Based on this analysis of firewall types, we created a model of an application-layer filtering PEP. This model is built according to the IETF architecture in [10] and extends the Liu's model of stateful firewalls to the application level. Fig. 4 presents a functional view of the model, composed of several modules. It is worth noting that existing firewalls use complex architectures optimized for performance, thus they do not necessarily use separated modules to implement the functionalities presented here.

The packets that arrive at the network interface are temporarily stored in the Input Unit, equipped with a buffer to reorder and reconstruct an entire PDU.

The PDU is then examined by the stateful section which is a modular entity: each component manages the state machine of a specific protocol, or handles other state information used by stateful and deep inspection algorithms. In particular, Figure 4 highlights the *TCP module*, which manages all the data associated to TCP connections (established, related, or the number of open connections, ...), the *Rule Hit module*, which maintains the hit counters associated to the rules in the stateless ruleset, two examples of *application protocol state managers* (for FTP and HTTP), and the *Proxy module* to maintain state info about authenticated users.

The stateful section accumulates state information into the Tag and updates the state table according to stateful rules. PDU and Tag are used by the PDP to identify the matching rules (whose conditions contain predicates on both the PDU and the state information). The PDP then notifies its decision to the Enforcement Unit that permits or denies the traffic flow.

IV. Squid access control features

We prove the effectiveness of our model by showing how it can be used to ease the specification and to perform the anomaly analysis of the Squid access control policies [9]. Therefore, we briefly present here this access control model to derive our requirements.

The open-source caching proxy Squid is widely used due to its great flexibility; it offers access control mechanisms

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TNET.2013.2293625

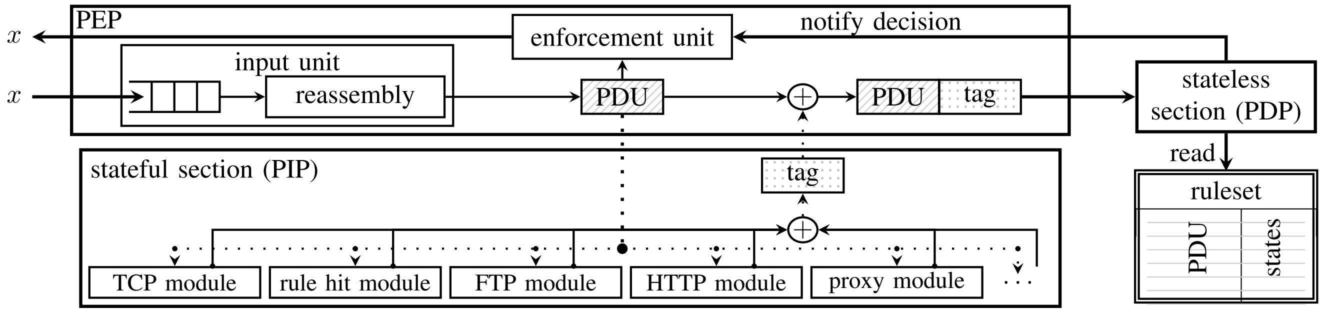


Fig. 4: Functional model of the PEP.

at different levels, from network data (e.g. MAC and IP addresses) to application information (e.g. host domain or browser information). The access control policy is written as a set of rules in the *squid.conf* file.

Three Squid commands are of our interest: *acl*, used to specify different types of conditions, *http_access* and *http_reply_access*, used to combine *acl*s for filtering rules. The *acl* syntax is:

```
acl aclname acltype value1 value2 ...
```

The string *aclname* univocally identifies a condition, *acltype* specifies the part of the packet considered by the matching process and the remaining fields define the values for which the condition is true. For instance, the following condition:

```
acl webPort port 80 443
```

is true if the destination port field in the packet header is either 80 or 443 (independent of the transport protocol).

When two *acl*s have the same name and type, Squid applies the logical disjunction. For instance, the following *acl*s are equivalent to the previous one:

```
acl webPort port 80
acl webPort port 443
```

The *http_access* syntax is:

```
http_access ("allow" | "deny") [acl1][acl2]...
http_reply_access ("allow"|"deny") [acl1][acl2]...
```

The *http_reply_access* command is used to filter replies to client requests, and it is complementary to *http_access*.

The Squid policy enforcement process supports only two actions, allow and deny, specified as first parameter, followed by a list of *acl*s. All the listed *acl*s are logically conjuncted when performing the matching process, that is, a rule is activated only when all its *acl*s match the PDU.

Squid applies the action from the first *http_access* command found in *squid.conf* which is true, that is, it uses the FMR resolution strategy. The default action is determined in a very peculiar way: if the action enforced by the last rule is Deny, then the default action is Allow, otherwise it is Deny. For this reason, to avoid unexpected behaviours, the best practice strongly recommends to add a “deny all” rule as last command.

Mapping Squid policies to our model requires the identification of the available actions and the selectors forming the decision space. First, as the actions considered by Squid are Allow and Deny, our action set is $\mathcal{A} = \{A, D\}$. Finding the selectors that must be included into the decision space is more complex and requires the analysis of the different *acltypes*. For most *acltypes* there is no ambiguity in defining the corresponding selector. For instance, it is easy to identify the selector type for addresses and ports (i.e. range-based) and the set of admissible values, that are analogous to the five-tuple example. However, three major differences with the stateless packet-filter case exist, because application-layer policies use:

- stateful conditions, not only on TCP connections but also at higher level (e.g. the number of user connections);
- regular expressions to define conditions on text fields, like URL and MIME type;
- alternate ways to define conditions on the same field, like path and domain conditions on URL.

A. Stateful conditions

Squid support various stateful conditions [13]. For example, it is possible to define the maximum number of HTTP connections open from the same IP address, via the *maxconn acl*, as in the following example that permits a maximum of four HTTP connections:

```
acl OverConnLimit maxconn 4
http_access deny OverConnLimit
```

Adding support to this condition type in our model is easy, because it can be mapped to a range-based selector. Therefore we just need to add the *conn* selector to the selection space. Practically, we can assume that the stateful section adds the user’s connections counter to the tag, hence the previous rule matches when it has a value greater than four. This is expressed in our model as:

```
((ips = any, ipd = any, ps = any,
  pd = any, proto = any, conn > 4), D)
```

Analogously, we can model the *max_ip_conn acl* (i.e. the maximum number of IP addresses from which the same user can connect) by introducing the *authIPno* range-based selector. In general, supporting stateful conditions requires just introducing additional exact-match and range-based selectors in the selection space. In fact, according to the model in Fig. 4, conditions on (dynamically maintained) stateful data are mapped to stateless conditions on tags.

B. Regular expressions

In other cases, mapping *acltypes* to selectors requires changes to the model. For instance, Squid supports filtering based on URLs, that are character strings with the following structure [14]:

scheme://<*host*>:<*port*>/<*path*>?<*searchpart*>

The following rule grants access to a web site whose domain ends with the string ".site.com":

```
acl dom1 dstdomain .site.com
http_access allow dom1
```

The condition in this rule cannot be mapped onto the geometric model as it supports only the exact match, prefix match and range-based selectors.

Some *acltypes* can be specified using regular expressions, as they are a very effective way to write complex conditions that match character strings, e.g. this rule allows access only to .htm pages within the .web.com domain:

```
acl url1 url_regex (.*).web.com/(.*).htm
http_access allow url1
```

Also this rule cannot be represented in the geometric model. Therefore, we conclude that a new selector type is needed to support conditions on strings: the *regex selector*. String matching is a subcase of regular expression matching. Therefore, from the theoretical point of view, both cases can be mapped to regex selectors. However, it is quite frequent to search for plain-text data set, thus implementations should consider optimized versions that internally distinguish the two cases.

It is worth noting that the analysis of Squid filtering capability produces requirements that also apply to other HTTP filters (like Apache with mod_proxy) as well as other protocols. For instance, Microsoft IIS implements FTP filtering capabilities and uses the <requestFiltering> command that introduces the need for regular expressions as it may discard messages based on URLs (e.g. <denyUrlSequences>) and for stateful conditions (e.g. <requestLimits>).

C. Overlapping condition types

As a last example, Squid offers four *acltypes* referring to URLs: *dstdomain* and *dstdomain_regex* express conditions on the <*host*> portion of the URL, *urlpath_regex* on the <*path*>?<*searchpart*> part, while *url_regex* refers to the entire URL. This means that conditions specified using these *acltypes* may intersect since their matching spaces are not disjoint. For instance, this is a set of intersecting *acl* directives:

```
acl dom1 dstdomain .site.com
acl dom2 dstdom_regex site
acl url1 url_regex (.*).web.com/(.*).htm
acl url2 urlpath_regex page.(.*)
```

It is evident that the *acl*s *dom1* and *dom2* may match the same URL, for instance http://site.com/page.asp. Analogously, both *url1* and *url2* match the URL http://www.web.com/page.htm. Moreover, *url2* matches some URLs matched also by *dom1* and *dom2*, such as URL http://site.com/page.asp.

In our model of Squid, all these *acl*s are associated to the same selector: the *url* selector. From the implementation point of view, it is better to minimize the use of regular expressions for improving the performance of the analysis and to produce optimal anomaly-free Squid configurations. For example, when a *dstdomain acl* (string matching) is intersected with a *dstdomain_regex* one (regex matching), the result should be represented as a *dstdomain*, since their intersection is either the entire *dstdomain acl* or the empty set. Additionally, *url_regex* matching is interpreted as *dstdomain_regex* or *urlpath_regex*, if they only apply to one part.

The rules presented above are thus translated to:

```
((ips = any, ipd = any, ps = any, pd = any,
  proto = any, conn = any, url = .site.com/), A)

((ips = any, ipd = any, ps = any, pd = any,
  proto = any, conn = any,
  url = (.*).web.com/(.*).htm), A)
```

The same consideration applies to other *acl*s pairs, namely *proxy_auth* and *proxy_auth_regex*, *srcdomain* and *srcdom_regex*, and *ident* and *ident_regex*.

V. REGEX SELECTORS AND ANOMALY ANALYSIS

We have seen in the previous section that *regex selectors* are needed to handle application-level policies, in addition to exact match, prefix match, and range-based selectors that already exist in packet filters. As a consequence, we need also to define methods to perform set operation, rule pair analysis, and general (multi-rule) anomaly analysis over regex selectors.

Intersecting two regular expressions is a complex operation. To simplify it, we translate regular expressions to deterministic automata – given their equivalence [15] – and operate on the latter. In fact, automata intersection is a well-known and (relatively) simple operation for which algorithms and implementations exist for several programming languages. Additionally, implementations exist to map regular expressions to automata. For example, Figure 5 depicts the intersection of two simple regular expressions performed using automata.

Unfortunately, the conversion from automata to regular expressions is no easy task. Different methods are available in literature, the most used ones being the transitive closure [16], the algebraic approach (Brzozowski's method) [17], and the state removal [18]. It is worth mentioning that this conversion is rarely needed and only for visualization purposes. The transitive closure approach has a simple implementation, but tends to create long regular expressions. The algebraic approach leans toward a recursive approach, which generates reasonably compact regular expressions, but its implementation using standard programming languages (e.g. Java, C) is too long and

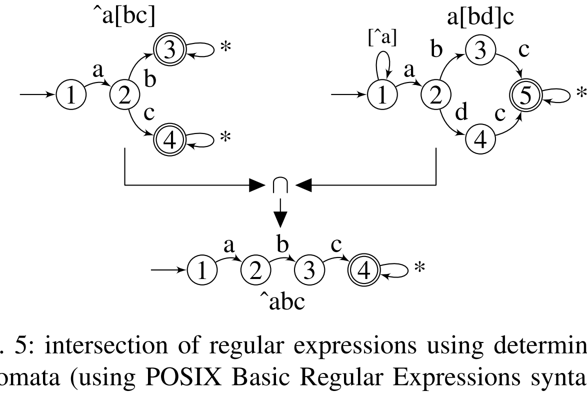


Fig. 5: intersection of regular expressions using deterministic automata (using POSIX Basic Regular Expressions syntax).

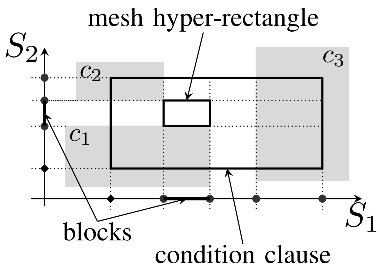


Fig. 6: Example of a lattice mesh partitioning a condition clause.

complex for the purpose of our prototype. Thereby, we adopted the state removal approach since it requires little effort to be adapted to our model.

It should be noted that, having introduced set operations on regex selectors, we can perform rule-pair analysis for policies expressed with regular expressions. In turn, this permits the identification of all the Al-Shaer anomalies for these policies.

We have now all the components to define an algorithm for general anomaly analysis with regex selectors. It has to handle the following task: given a policy $(R, \mathfrak{R}, E, d)$ and a target rule $r = (c, a)$, the algorithm verifies if r is *unnecessary in* R, that is if the policy is unchanged when r is removed from R. Intuitively, the target rule is unnecessary only if:

- r is completely "overridden" by other rules at higher priority, or
- r is not completely overridden but it does neither "override" any rule at lower priority nor overrides parts where the policy would apply the default action.

The "sub-optimality property" serves to this purpose, verifying if removing the rule changes the action the policy would return in any of the points of the decision space by checking the following property

$$\mathfrak{R}(\mathrm{match}_R(x)) = \mathfrak{R}(\mathrm{match}_R(x) \setminus \{r\}) \tag{1}$$

The algorithms presented in [6] are based on the computation of the eff_p function that returns the portion of the target rule not hidden. To avoid set minus operations, eff_p is calculated by intersecting rules in canonical form. This method is not suitable for application layer policies due to the high number of selectors. In fact, representing the set minus of two m-dimensional hyper-rectangles may require up to m hyper-rectangles, and an equivalent number of rules intersections. Moreover, the number of rule operations exponentially increases with the number of input hyper-rectangles.

Thus we introduce a new approach, that avoids rule operations. We will assume in this section the following naming convention. $c = s_1 \times \cdots \times s_m$, is the condition clause of r formed by m selectors, and s_i is the condition of the i-th selector S_i of c. The other rules in $R \setminus \{r\}$ will be identified as $r_j = (c_j, a_j)$ with $c_j = s_{j,1} \times \cdots \times s_{j,m}$, thus $s_{j,i}$ is the condition of c_j in S_i.

The idea behind this approach is to create a lattice mesh that partitions c_i in hyper-rectangles where $\mathrm{match}_R(x)$ is constant so that the "sub-optimality property" 1 can be evaluated without explicitly calculating eff_p. An example of a mesh that partitions a condition clause is presented in Fig. 6. Note that Fig. 6, 7a, 8a, 8b, and 8c present cases for range-based selectors for graphical immediateness only. However, as it will be evident in this section, the verification algorithm actually works regardless of the selector type.

To build the mesh, each selector s_i is split into *blocks*. Formally, a block is defined as

$$b \subseteq s_i \text{ such that } \forall x \in b, \mathrm{match}_R(x) \subseteq R$$

A set of blocks $B_i = \{b_{i,k}\}_k$ exists that partitions s_i, that is:

$$s_i = \bigcup_{i \in B_i} b_{i,k} \text{ and } b_{i,k_1} \cap b_{i,k_2} = \emptyset \text{ with } k_1 \neq k_2$$

Each block is associated to matching rules via the ρ function:

$$\begin{array}{rccl} \rho: & \bigcup_i B_i & \longrightarrow & 2^R \\ & b_{i,j} & \longmapsto & \mathrm{match}_R(x) \subseteq R,\ x \in b_{i,j} \end{array}$$

The *blocks identification problem* is the following one: given $s_i \subseteq S_i$ and the n selectors $s_{j,i}$, find the minimum number of blocks $B_i = \{b_{i,k}\}_k$ that partition s_i. This implies determining blocks with distinct ρ values, i.e. $b_{i,k_1} \neq b_{i,k_2}$ when $k_1 \neq k_2$.

A *c-mesh hyper-rectangle* is defined as $h = w_1 \times \cdots \times w_m \subseteq c$ with $w_i \subseteq s_i$ such that $\forall x \in h, \mathrm{match}_R(x) \subseteq R$. It is also possible to find a set of mesh hyper-rectangles that partition c, that is:

$$c = \bigcup_i h_i \text{ and } h_{i_1} \cap h_{i_2} = \emptyset \text{ with } i_1 \neq i_2$$

A set H_c of mesh hyper-rectangles that partition c can be obtained as Cartesian products of blocks, that is:

$$H_c = \{b_{1,i_1} \times b_{2,i_2} \times \cdots \times b_{m,i_m}\} \quad b_{1,i_1} \in B_i, \ldots, b_{1,i_m} \in B_m$$

The matching rules in $h = b_{1,i_1} \times \cdots \times b_{m,i_m}$ can be calculated by extending the ρ function to:

$$\rho(h) = \bigcap_j \rho(b_{j,i_j})$$

because the matching rules in a mesh hyper-rectangle also match all its constituting blocks.

Working with H_C mesh hyper-rectangles, the *general criterion* to determine "not unnecessary" rules becomes:

$$\exists h \in H_c,\ \mathfrak{R}(\rho(h)) \neq \mathfrak{R}(\rho(h) \setminus \{r\}) \tag{2}$$

This states that the target rule is necessary if property 1 is not verified for at least one mesh hyper-rectangle.

From property 2 we derive the *block criterion* to identify not unnecessary rules based on block information only. If, regardless of the selector, there exists a block b such that $\rho(b) = \emptyset$ and a (the action enforced by the target rule) is different from the default action d, then the rule is not unnecessary. In fact, each mesh hyper-rectangle h formed by using b will also have $\rho(h) = \emptyset$, and $\mathfrak{R}\{\rho(h))\} = \mathfrak{R}\{\emptyset\} = d \neq \mathfrak{R}\{r\} = a$, thus property 1 is not satisfied.

The rest of this section is devoted to algorithmically presenting the general anomaly analysis algorithm and its complexity.

Algorithm 1 presents the procedure that determines if a rule is unnecessary. We assume that the rule pair analysis is completed (as it is computationally faster) and the identified redundant and shadowed rules have been removed. Moreover, we assume that the intersecting rules have been determined. After an initialization phase in which the blocks are computed (BUILDBLOCKS, line 1), two verifications are performed: a first selector-wise verification, SELECTORVERIFY (line 2), implements the block criterion, while the second one, MESHVERIFY (line 3), implements the general criterion used if SELECTORVERIFY is unable to reach a verdict on the rule necessity.

Algorithm 1 UNNECESSARYVERIFY(r, R)

Input: $r \in R$, the rule to analyze
Input: R, the policy ruleset
Output: Boolean, True if r is unnecessary in R
1: $B \leftarrow$BUILDBLOCKS(r, R)
2: **if** SELECTORVERIFY(B) does not recognise r as not unnecessary **then**
3: execute MESHVERIFY$(B, 1)$
4: **end if**

The blocks identification problem is implemented by BUILDBLOCKS (Algorithm 2) according to the selector type. It uses BUILDBLOCKSREGEX (line 3) for regex and BUILDBLOCKSRANGE (line 6) for other selectors. A further step is required to process unordered exact match selectors as ordered ranges (line 5). This is easily performed by mapping every point in an exact match selector to an integer with standard ordering. This task is very natural, as packet header fields use bits (i.e. integers) to encode information, like protocol type in IP packets.

Algorithm 2 BUILDBLOCKS(r, R)

Input: $r \in R$ the rule to analyze
Input: R the policy ruleset
Output: $B = \{B_i\}_i$, an array of m blocks, one for each selector
1: **for all** selector $s_i \subseteq S_i$ in c **do**
2: **if** S_i is a regex selector **then**
3: $B_i \leftarrow$BUILDBLOCKSREGEX(r, R, i)
4: **else**
5: interpret S_i as a range-based selector
6: $B_i \leftarrow$BUILDBLOCKSRANGE(r, R, i)
7: **end if**
8: **end for**
9: **return** B

Algorithm 3 presents block identification for range-based selectors. We initially assume, for ease of presentation, that conditions are formed by a single range, point or regular expression, and extend later the results to the general case. BUILDBLOCKSRANGE identifies the blocks by splitting the initial condition s_i. Every time a new condition $s_{j,i}$ is considered, s_i is split in at most two points, that is, if $s_i \cap s_{j,i} = [x_j, y_j]$ at x_j (line 6) and $y_j + 1$ (line 7). Fig.7a depicts how the initial condition is split when adding a new condition and how ρ is assigned to blocks.

The following theorem holds for range-based selectors.

Algorithm 3 BUILDBLOCKSRANGE(r, R, i)

Input: R the policy ruleset
Input: $r = (c, a) \in R$ the rule to analyse
Input: i the ordinal of the range-based selector in $\mathcal{S}$
Output: the blocks set B_i
1: create an empty blocks set B_i and add s_i
2: set $\rho(s_i) \leftarrow \emptyset$
3: **for all** intersecting rules $r_j = (c_j, a_j)$ **do**
4: $s_{j,i} \leftarrow$ the i-th selector of c_j
5: calculate $\nu_\cap = [x_j, y_j] \leftarrow s_i \cap s_{j,i}$
6: take the block $\nu = [\nu_s, \nu_e]$ such that $x_j \in \nu$, remove it from B_i, split it in $[\nu_s, x_j - 1]$ and $[x_j, \nu_e]$ and insert the results in B_i
7: take the block $\mu = [\mu_s, \mu_e]$ such that $y_j + 1 \in \mu$, remove it from B_i, split it in $[\mu_s, y_j]$ and $[y_j + 1, \mu_e]$ and insert the results in B_i
8: **for each** range $\tau \subseteq [x_j, y_j]$ in B_i **do**
9: $\rho(\tau) \leftarrow \rho(\tau) \cup r_j$
10: **end for**
11: **end for**

Theorem 1: The number of block induced in $s_i = [x, y]$ by n_r conditions $\{s_{i,j}\}$ (with $s_{i,j} = [x_j, y_j]$, $x \le x_j \le y$ and $x \le y_j \le y$) is at most $2n_r$.
This theorem states that the number of blocks is at most twice the number of intersecting ranges, that in this simple case is also the number of rules intersecting r. Let us prove it using a constructive proof. We build the set of integers $P = \{x, y+1\} \cup \{x_j\}_j \cup \{y_j + 1\}$, that is, the set composed of all the starting points and the successors of all endpoints. By ordering and indexing the points in P according to their value (i.e. $p_i, p_j \in P \Leftrightarrow i < j$), we can describe a set of ranges that partition s_i, that is

$$s_i = [p_1, p_2 - 1] \cup [p_2, p_3 - 1] \cup \cdots \cup [p_{|P|-1}, p_{|P|} - 1]$$

where $p_1 = x$ and $p_{|P|} = y + 1$. The range number is $|P| - 1$.

If all the integers in P are distinct, $|P| = 2n_r + 2$ holds, and there are $2n_r + 1$ ranges. This number is maximal. In that case, the initial range $\nu_1 = [x, p_1 - 1]$ and the final range $\nu_2 = [p_{|P|-1}, y]$ have $\rho(\nu_1) = \rho(\nu_2) = \emptyset$. Therefore, they can be merged and we have at most $2n_r$ blocks. To finish the proof we prove that there exists a case with $2n_r$ blocks. If we have as input conditions $s_{i,j} = [x_j, y_j]$ with $x < x_1 < x_2 < y_1$, and for all j, $x_{j+2} = y_j + 1$ and $x_{j-1} < y_j < x_j$ hold, then the $2n_r$ generated blocks with distinct ρ are:

- $\rho([x, x_1]) = \emptyset$,
- $\rho(x_1, x_2]) = \{r_1\}$,
- $\rho([x_j, y_{j-1}]) = \{r_{j-1}, r_j\}$ with $j \in [3, 2n_r - 2]$
- $\rho([y_{n_r - 1}, y_{n_r}]) = \{r_{n_r}\}$, and
- $\rho([y_{n_i}, y]) = \emptyset$.

In two other cases we have a maximal number of blocks, that is $2n_r$ blocks, if one of the y_i equals y and if one of the x_i equals x. In all other cases $|P| \le 2n_r$.

The analysis can be extended to conditions described as union of ranges and it is easy to show that, if conditions are formed by $\overline{K}$ ranges, the points are at most $|P| \le 2\overline{K}n_r$

and the number of blocks $|B_i| \leq \min\{2^{n_r}, 2\overline{K}n_r\}$. While for $n_r \leq 4$ the 2^{n_r} factor prevails, for larger n_r (the cases interesting for the complexity analysis) the linear term prevails.

Algorithm 4 BUILDBLOCKSREGEX(r, R, i)

Input: R the policy ruleset
Input: $r = (c, a) \in R$ the rule to analyse
Input: i the ordinal of the regex selector in $\mathcal{S}$
Output: the blocks set B_i

1: create a list of blocks B_i and add s_i
2: set $\rho(s_i) \leftarrow \emptyset$
3: **for all** intersecting rules $r_j = (c_j, a_j)$ **do**
4:     $s_{j,i} \leftarrow$ the i-th selector of c_j
5:     $x \leftarrow s_i \cap s_{j,i} \neq \emptyset$
6:     $y \leftarrow s_i \cap \neg s_{j,i}$
7:     **for each** block b in B_i **do**
8:         **if** $x \leftarrow b \cap x$ **then** ▷ $\neq \emptyset$ because r_j intersects r
9:             remove b and add x in B_i
10:             $\rho(x) \leftarrow \rho(l) \cup \{r_j\}$
11:         **end if**
12:         **if** $y \leftarrow b \cap y \neq \emptyset$ **then**
13:             remove b (if still in B_i) and add y in B_i
14:             $\rho(y) \leftarrow \rho(b)$
15:         **end if**
16:     **end for**
17: **end for**

On another hand, when working with regex selectors, the ordering cannot be used to determine the blocks that need to be enumerated explicitly (Algorithm 4). In fact, every time a new rule is considered, the algorithm first calculates the intersection x between the root node and the selector $s_{j,i}$. If x is not empty (line 5) it also calculates the intersection y with the negation of $s_{j,i}$ (line 6). Then, for each previously computed block b in B_i, it calculates the intersection with x (line 8) and y (line 12), and if they are not empty, it substitutes b and updates ρ for x and y (lines 9-10 and 13-14). For instance, Fig. 7b displays the blocks generated from the condition s_i when intersected with the regex conditions $s_{i,1}$, $s_{i,2}$, $s_{i,3}$ whose Venn diagram is shown in Fig. 7b. The figure presents the intermediate algorithm iterations in form of a tree. The tree helps us to quantify the maximum number of blocks that can be formed in a regex selector, that is, 2^{n_r}, where n_r is the number of rules that intersect r.

Algorithm 5 shows the implementation of the block criterion used for selector-wise verification that simply consists in checking the ρ value for all blocks.

Algorithm 5 SELECTORVERIFY(B)

Input: B the set of the selector blocks s_i
Output: Boolean, True if the rule is not selector-wise hidden

for each data structure B_i in B **do**
    for all blocks b in B_i **do**
        if $\rho(b) = \emptyset$ and $a \neq d$ **then** r is necessary
        end if
    end for
end for

Graphically, the meaning of this approach is more evident. For example, Fig. 8a presents the block $b_{1,3}$ having empty ρ, that corresponds to a "white slot" that ranges from the beginning to the end of the other selector. If r is dropped, in that slot the default action is applied. This algorithm can be further optimized. In fact, in a regex selector, an empty block may appear only as intersection of negated conditions $\neg s_{j,i}$. In Fig. 7c, the empty block is $b_{1,5} = s_i \cap \neg s_{1,i} \cap \neg s_{2,i} \cap \neg s_{3,i} \cap$, that is, the "rightmost leaf node".

However, if all the selectors are partitioned in blocks with non-empty ρ, it does not means that the rule is unnecessary. Fig. 8b presents a rule for which the block criterion is satisfied that may be unnecessary if $a \neq d$. The MESHVERIFY procedure is used in these cases (Algorithm 6).

Algorithm 6 MESHVERIFY(B, i)

Input: $B = \{B_i\}_i$, an array of m blocks sets
Input: i the current selector

1: assume r unnecessary and redundant
2: **for all** blocks b in B_i **do**
3:     $\rho(h) \leftarrow \rho(h) \cap \rho(b)$
4:     **if** $\rho(h) = \emptyset$ and $a \neq d$ **then** r is necessary
5:     **end if**
6:     **if** i=m **then** ▷ last selector
7:         **if** $\Re\{\rho(h)\} \neq \Re\{\rho(h) \cup \{r\}\}$ **then**
8:             r is necessary
9:         **else**
10:             **if** $\Re\{\rho(h)\} \neq a$ **then**
11:                 if unnecessary, assume r shadowed
12:             **end if**
13:         **end if**
14:     **else** ▷ intermediate selectors
15:         continue recursively MESHVERIFY$(B, i + 1, m)$
16:     **end if**
17: **end for**

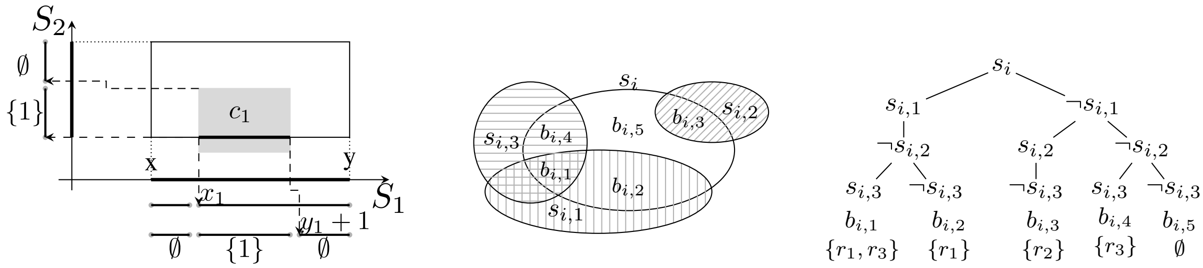
MESHVERIFY follows this approach: it assumes that r is unnecessary if the contrary is not proven. It recursively obtains all H_c mesh hyper-rectangles using the previously computed blocks and updates ρ (line 3). If ρ becomes empty and if $a \neq d$, then r is declared necessary (line 4) without the need to complete the recursion. This is the case in Fig.8b, where ρ in $b_{1,2} \times b_{2,2}$ is empty.

At the last selector, when mesh hyper-rectangles are complete, the algorithm checks if the property 1 holds. According to the general criterion, r is declared necessary as soon as the algorithm finds a hyper-rectangle such that the property 1 does not hold. If no mesh hyper-rectangle contradicts property 1 the rule is unnecessary. An unnecessary rule is redundant if its action is not overridden in any of the mesh hyper-rectangles (line 11) otherwise it is shadowed. This is the case in Fig. 8c, where the verification must check property 1 for all the 16 hyper-rectangles to declare it unnecessary.

VI. COMPUTATIONAL ANALYSIS

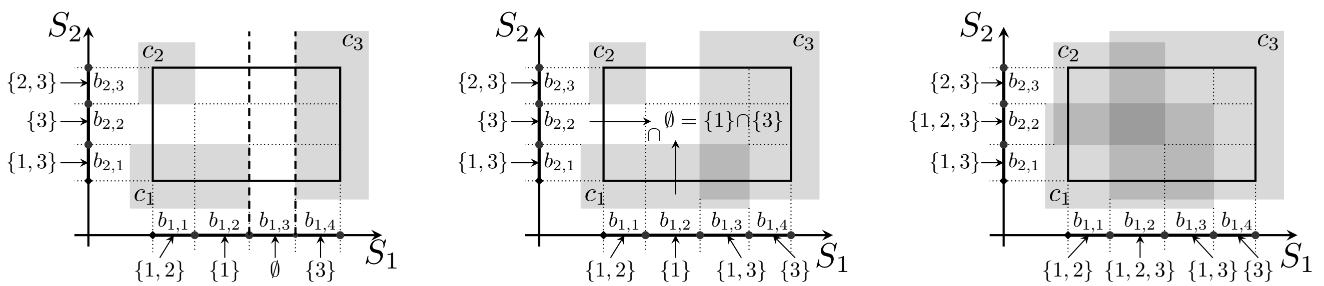
We proceed now to the computational analysis of our algorithm for general anomaly identification for policies using

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TNET.2013.2293625



(a) Block splitting for range-based selectors. (b) Regex selector: Venn diagram. (c) Regex selector: tree representation.

Fig. 7: Example of block calculation for different selector types.



(a) A rule whose necessity is verified with SELECTORVERIFY. (b) A rule with a mesh hyper-rectangle with empty ρ (MESHVERIFY). (c) A rule whose necessity needs to be verified using MESHVERIFY.

Fig. 8: Example of unnecessary rules verification.

regex selectors too. To this purpose, we will use the following symbols:

- (variables) $n = n_r + \overline{n_r}$ is the number of rules in the ruleset R, among them only n_r intersect the target rule r;
- (constants) $m = m_R + m_E$ is the number of selectors in the decision space of rules in R, m_R of range-based type and m_E of regex type; m_R and m_E are fixed values that depend on the policy. Since the general anomaly analysis maps exact-match selectors to range-based ones, m_R is actually the sum of non-regex selectors;
- (computational costs) I_R and I_E are respectively the cost to intersect two conditions in range-based and regex selectors; N is the cost to add a new block in a range-based selector and T in a regex one and to update ρ; ϵ is the cost of other operations, like comparing ρ values, extracting a selector from a rule, or a combination of them.

The worst case scenario happens when the target rule is unnecessary, because in all other cases the verification stops before performing all steps. In that case, the complexity of UNNECESSARYVERIFY is the sum of the computational costs of BUILDBLOCKS, SELECTORVERIFY, and MESHVERIFY.

Assessing BUILDBLOCKS complexity requires to consider both BUILDBLOCKSREGEX and BUILDBLOCKSRANGE cases. BUILDBLOCKSREGEX has worst case complexity given by the following formula:

$$n_r(\epsilon + 2I_E) + 2\left(I_E + \epsilon\right)\sum_{k=1}^{n_r}\left|B_i^{(k)}\right|$$

This is because instructions at lines 4-6 are executed n_r times, and the inner cycle at line 7 requires in the worst case two intersections, two updates and three list operations for each of the blocks already in B_i. At the iteration k, the blocks in $B_i^{(k)}$ are at most 2^k, thus, the following holds $\sum_{k=1}^{n_r}\left|B_i^{(k)}\right| = \sum_{k=1}^{n_r} 2^k = 2(2_i^n - 1)$. The complexity of this function is $O\left(2^{n_r}\right)$ because previous formula becomes $n_r(\epsilon + 2I_E) + 4\left(I_E + \epsilon\right)\left(2^{n_r} - 1\right)$.

BUILDBLOCKSRANGE has worst case complexity given by the following formula:

$$n_r\left(\epsilon + I_R + 2N\right) + \sum_{k=1}^{n_r}\left|B_i^{(k)}\right|\epsilon$$

This is because lines 4-7 are executed n_r times, and the inner cycle requires in the worst case one ρ update for each of the blocks already in B_i. At the iteration k, the blocks are at most $B_i^{(k)} \leq 2\overline{K}k$, thus, $\sum_{k=1}^{n_r}\left|B_i^{(k)}\right| = \sum_{k=1}^{n_r}\overline{K}k = \overline{K}n_r + \overline{K}n_r^2$. Therefore the complexity is $O\left(n_r^2\right)$ because the previous formula becomes $\left(\epsilon + I_R + 2N + \overline{K}\epsilon\right)n_r + \overline{K}\epsilon n_r^2$.

We can conclude that overall the worst case complexity of BUILDBLOCKS is $O(2^{n_r})$.

On the other hand, the actual verifications only depend on the number of blocks. SELECTORVERIFY checks the value of ρ once for each block regardless of the selector, that is, the complexity is $O\left(2^{n_r}\right)$, in fact:

$$\sum_{i=1}^{m}|B_i|\epsilon = (m_R(2\overline{K}n_r) + m_E(2^{n_r}))\epsilon$$

MESHVERIFY intersects ρ values and compares two resolution strategy results for each mesh hyper-rectangle:

$$\prod_i |B_i|\epsilon = (2\overline{K}n_r)^{m_R}(2^{n_r})^{m_E}\epsilon = (\overline{K}n_r)^{m_R}2^{n_r m_E + m_R}$$

that is $\sim O\left(n_r^{m_R} 2^{n_r}\right)$. It is evident, that the computational cost of VERIFYUNNECESSARY is dominated by MESHVERIFY.

A discussion is needed to explain why, although the worst case is very bad, this approach works well in practice (as experimentally verified in section VI).

The parameter that directly affects performance is the number $n_r \leq n$ of rules intersecting the target rule r, that is bounded by the rule set size n. Also m_R and m_E affect the performance but they are not variable.

In real rule sets, n_r is very small and is practically independent of the rule set size n. Theoretically, rules should intersect only in case of exceptions/generalizations (e.g. rules to express policies like "all the IP addresses of a subnet but one are allowed to reach a service"). Administrators avoid or limit intersecting rules by logically partitioning the condition space starting from one or a few selectors. A typical example is writing rules according to the subnets (IP source or destination), then by ports (i.e. services).

For packet filters, statistics are available to estimate ruleset size and number of intersecting rules. Packet filtering policies may have thousands rules, the biggest rule set analysed by Wool [19] being of 7400 rules. According to the data in [20], the maximum number of intersecting rules in the analysed stateless firewall is 4 , and the maximum number of intersecting conditions is 5, regardless of the rule set size. The work [4] helps to quantify rules that do not intersect at all: it reports that in the worst case of inexperienced administrators, about 9% of the rules are correlated (i.e. intersect at least another rule). This in turn means that 91% of the rules are not overlapping at all if pair-wise redundant or shadowed rules are removed.

However, statistical data are not available for application firewalls. Application firewall rulesets may have the same size as stateless ones. In fact, if used as reverse proxy, application firewalls are placed very close to the protected service (e.g. HTTP server, web service), serve a limited number of IP addresses and have few rules, but if used as forward proxy they may contain several rules. In this case, rules are often partitioned by destination URL. A further analysis has been done to verify if the same considerations apply to application firewalls. We considered 15 anonymous or publicly available Squid configuration files composed by 20 to 50 rules and we analysed the correlation among conditions and among rules. We verified that Taylor's results are compatible with the application layer scenario with minor differences. The most important one concerns conditions on URLs, where the number of intersecting conditions is greater than five, especially because of an inaccurate use of wildcards[1]. The worst case we examined had 7 intersecting nested URL conditions and 5 intersecting rules.

Both SELECTORVERIFY and MESHVERIFY complexity depend on the cardinality of block sets B_i, that in turn depends on n_r. The very limited number of blocks is another reason for practical usability of our approach. In fact conditions are not equally distributed on the whole selector but clustered. Conditions on source and destination IP addresses correspond to subnets or single IP addresses. Therefore, the number of blocks is less than the worst case because endpoints are not distinct. Moreover, it is not possible to have more blocks than points in the condition, e.g. a condition including one IP address forms exactly one block. Source ports in most of the cases are left unspecified or exclude the well known ports (e.g. 1024–65535), while destination ports are clustered on the most used services (e.g. 80, 22, 443). URLs are very often organized by destination domain or hierarchically organized by domain/URL path so that conditions are nested or disjoint. This guarantees that the worst case 2^{n_r} for regex selectors is rarely approached, if ever. The analysis of Squid rulesets produced one hundred mesh hyper-rectangles in the worst case. We noticed that even tough dozens of selector types are available for application firewalls, policy writers tend to use only a bunch of them for each rule: we noticed that no more than five selectors are specified, with the unspecified ones working as wildcards. This also strongly limits the number and product of block sets size.

Finally, the average number of ranges per condition is 1 for some specification languages that do not allow union of ranges and, in general, union of ranges is not abused. In most cases $\overline{K}$ is a number close to 1.

[1]For example, the URL conditions "*site1.com*" and "*site2.com*" are interpreted by Squid as "*.*site1.com.**" and "*.*site2.com.**" thus they actually intersect, e.g. the domain *www.site1.commercial.site2.com* matches both. This intersection might be avoided using end-of-line anchor, e.g.*site1.com$*

VII. IMPLEMENTATION

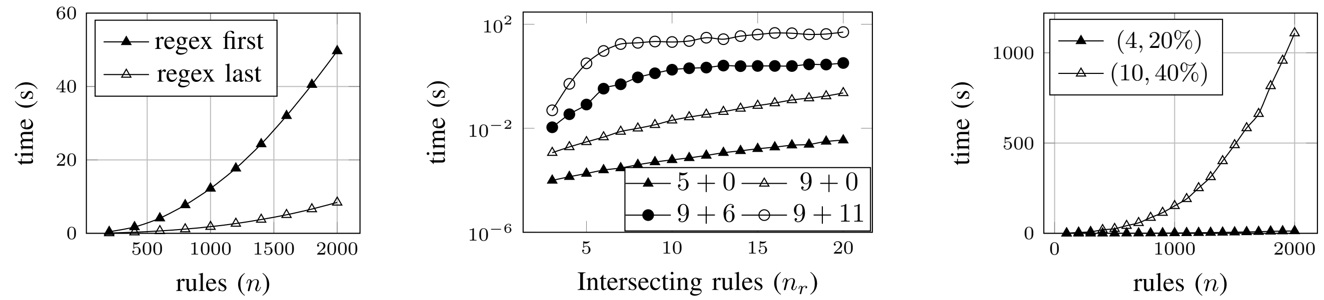
The tool presented in [6] supports policy specification with several resolution strategies and performs rule-pair and multi-rule analysis and policy translation. We extended it to support application firewalls and the Squid syntax. The anomaly detection is implemented in two steps, pairwise then multi-rule analysis. Detected anomalies are presented to administrators for validation purposes.

Range-based selectors are implemented as integers and set operations are optimized resorting to their natural order. Prefix match selectors are mapped to range-based selectors. Exact match selectors use bit sets, that is ordered strings of boolean digits. Each element is associated to a specific position in the bit set: if the element is present in the condition, then the corresponding bit is set. The intersection is mapped to the bitwise *AND* operation, the union to the bitwise *OR* and the set minus to the *AND-NOT*. We map regex selectors to automata by extending the *dk.brics.automaton* Java package from A. Møller [21], that offers translation of regular expressions to automata and provides some set operations among them. Conditions with string matching use regex selectors, taking advantage of the "singleton strings" feature of *dk.brics.automaton*. Furthermore, we implemented missing set operations and the algorithm to convert automata to regular expressions.

It is worth noting that the tool can be easily extended to support other rule types and scenarios other than Squid with little or no changes to the source code.

A. Performance analysis

To complete the assessment of the practical usability of our approach, an extensive testing of the anomaly detection



(a) Impact of selector order on condition clause intersection performance.

(b) Time to detect unnecessary rules depending on n_r for different rule types ($m_R + m_E$).

(c) Policy analysis time depending on the rule set size for policy with different $(n_r^{\max}, \sigma)$.

Fig. 9: Test results

process has been conducted. Tests were performed using a computer equipped with a Intel Core i7 (2.7 GHz) CPU and 8 GB RAM, running Java 1.7 on top of a Linux 6 OS.

The most interesting parameter for usability is the time required to analyse a policy. Testing on real policies is impractical as they are treated at the maximum confidentiality level, thus not freely available. The ones we were able to access are too small and too little to be statistically significant. Therefore, we tested the model against synthesized rulesets.

A full analysis includes an initial pairwise analysis and a general anomaly analysis. Pairwise analysis mainly depends on the efficiency of set operations and comparisons (subset, superset, equivalent, disjoint). Therefore, with respect to [6], there are two differences: there are more selectors (up to 20) and regex selectors are used.

To this purpose, we first evaluated the time to perform set operations within selectors. Table I reports the time to perform one million set intersections and unions on the three different selector types. Range-based and exact match are very efficient while regex selectors are considerably slower (approximately, $I_E \sim 100 I_R$).

	regex	*range-based*	*exact match*
union	5.43 s	0.091 s	0.033 s
intersection	7.54 s	0.054 s	0.067 s

TABLE I: Performance of operations over selector types.

Then we estimated the time to intersect condition clauses. It depends on the probability of intersection in each selector and on the number of regex selectors. As condition clauses are the Cartesian product of conditions, in our prototype the intersection works selector-wise and stops as soon as an empty intersection is found (and due to independence among selectors it can be also parallelized). Thus the average time is:

$$\begin{aligned}\bar{t} = \; & \eta_1 t_1 + (1-\eta_1)\, t_2 + (1-\eta_1)(1-\eta_2)\, t_3 + \\ & + \cdots + (1-\eta_1)(1-\eta_2)\cdots(1-\eta_{m-1})\, t_m\end{aligned}$$

where η_i is the probability that two conditions intersect in the selector S_i, and t_i is the time to perform the intersection in S_i. Thus the order of the selectors affects the average time. Figure 9a displays the time to perform the intersection depending on the ruleset cardinality when regex are used as first or last selectors (and $\eta_i = 30\%$ for each selector) for Squid rules, composed of 9 non-regex and 11 regex selectors. The overall performance drastically improves if the intersection between regex conditions is calculated later. Moreover, further improvements are expected if statistical data are used to determine the selector order.

The most significant evaluation concerns general anomaly analysis. Two measurements have been performed: the time to verify that a rule is unnecessary depending on the number of intersecting rules, and the time to perform general anomaly analysis on a rule set depending on the rule set size.

In the first test, we created ad hoc unnecessary rules and measured the time to perform the verification depending on n_r. We generated realistic range-based conditions, composed of at most 5 ranges with $\overline{K} = 3$, exact match conditions composed of randomly generated sets of points, and random regex conditions (at most $\overline{X}$ of them are intersecting but not nested). In fact, generating regular expressions such that every pair of them is intersecting but not nested is quite complex (e.g. URLs, browsers' names), thus we expect that the possibility for administrators to generate them accidentally is also small. No conditions use wildcards.

Fig. 9b presents the results for four sample rules types:

1) five-tuple rules ($m_R = 5, m_E = 0$),
2) Squid rules without regex selectors ($m_R = 9, m_E = 0$),
3) Squid rules without the uncommon regex selectors ($m_R = 9, m_E = 6$), (e.g. authentication, ident)
4) Squid rules.

In cases 1 and 2, that do not include regex selectors, the algorithm is able to determine that a rule is unnecessary in less than one second even when there are 20 intersecting rules (respectively 3.5 ms and 0.229 s on average). The time for verification grows less than exponentially with n_r but it may be intractable. However, even if this plot can be theoretically extended to $n_r \geq 20$, those cases are very unlikely to happen. Cases 3 and 4, that include regular expressions, initially grow faster but this trend decreases with n_r. This also depends on our decision to produce at most $\overline{X}$ intersecting but not nested regex conditions. Detecting unnecessary rules requires respectively 3.24 s and 59.47 s on average. The worst case we measured took about 600 s to identify an unnecessary rule covered by 12 rules.

For the second class of tests (the time to perform anomaly analysis, both rule pair and general one) we used two parameters to produce realistic policies: $n_r^{\max}$, the maximum number of rules intersecting simultaneously, and σ, the percentage of intersecting rules in the ruleset (discussed in section VI). The test was performed on two series of policies, randomly generated with $(n_r^{\max} = 4,\ \sigma = 20\%)$ and $(n_r^{\max} = 10,\ \sigma = 40\%)$. Results in Fig. 9c show that realistic policies can be analysed in a short time: management of very large correlated policies is compatible with normal administrator activity. We re-run the same test on new policies that use different resolution strategy and obtained the same results. We conclude that performance is independent of the resolution strategy.

VIII. Related work

Several works treat policy anomaly classification and detection. The concept of conflicts analysis has been initially introduced by Sloman for distributed system management with techniques to solve them [22] [23] [24]. However, these techniques are not directly applicable to firewall policies. Many seminal papers present solutions for the analysis of packet filtering. First works concentrated on efficient representations of the rulesets as conflicting rules decrease performance. Hazelhurst presented solutions based on binary decision diagrams (BDDs) [25], Hari [26] proposed the use of tries, Baboescu [27] the use of bit vectors, and Srinivasan [28] the Tuple Space Search classification algorithm.

Then the focus moved to approaches that query the firewall policy, like Fang [29], a simulation-based engine that performs simple query aggregation, and its successor Firewall Analyzer (formerly known as Lumeta) [30], [31]. Recently, Liu proposed a query engine and the Structured Firewall Query Language [32], that he applied to the analysis of corporate networks composed of packet filters and NATs [33]. None of these works consider stateful or application firewalls.

Other approaches proposed the exhaustive anomaly detection. Al-Shaer focused on the analysis of single packet filters [4], and on distributed firewalls [34]. His work has two main limitations: it considers only the packet filter scenario (i.e., stateful and application layer firewalls are not supported), and he detects only anomalies in rule pairs (i.e., anomalies that arise considering more rules are not considered). His classification is the starting point of several works that share the same limitations. Bouhoula [35] used rule field logical relations, Thanasegaran [36] bit vectors that allow the detection of rule pair anomalies more efficiently but fail to effectively express conditions on ordered fields (e.g. port numbers).

Anomaly detection has been addressed with different perspectives. The FIREMAN tool [37] uses BDDs to detect anomalies, and checks if a distributed policy complies with an end-to-end policy. In the field of ruleset optimization by redundancy removal, Gouda [38] and Liu [39] introduced techniques based on Firewall Decision Diagrams. Abedin proposed a real-time ruleset optimization approach based on data mining techniques [40]. Alfaro proposed a set of algorithms to remove anomalies between packet filters and NIDS in distributed systems [41], recently implemented in MIRAGE [42]. Hu [43] proposed to divide the five-tuple decision spaces into disjoint hyper-rectangles where conflicts are resolved using a combination of automatic strategies and manual administrator effort driven by risk analysis considerations. A completely different approach is presented by Bandara, that uses argumentation logic and achieves excellent performance [44], and Hu that introduced an ontology-based anomaly management framework that delegates set operation to BDDs [45].

Our work aims at detecting anomalies also for stateful and application-layer policies, it supports strategies other than FMR. Moreover, as it is based on the geometric model easily extends to other rule types. The query approach is completely different as it does not aim at finding all the inconsistencies and strongly relies on the selection of the proper queries, as "users often do not know what to query" [31]. Therefore, in our opinion, the impact of the human factor is only shifted, on the other hand, authors focussing on queries object that, anomaly analysis is impractical as too many anomalies may be detected to be manually processed [32].

Stateful firewall analysis is less addressed in literature. Besides the already discussed work of Liu [8], we mention Cuppens [46] that detects rules that do not allow the normal TCP setup and termination for allowed connections, or rules that block allowed related FTP connections. Our tool also identifies these anomalies, nevertheless, they do not appear in application layer protocols. Buttyán [47] stated that "stateful is not harder than stateless", but this is only partially sharable as their model simply adds one string field (treated as an exact match selector) to the FIREMAN five-tuple decision space. The stateful case is harder because there are new anomalies and it is computationally more complex.

IX. Conclusions

This paper presented a model for policy anomaly analysis in application firewalls and a tool implementing it. The proposed model is able to manage text-based content filtering specified with regular expressions. The model effectiveness has been successfully tested against the access control features of Squid, a well-known HTTP proxy. Together with the effectiveness, encouraging results come from the performance analysis. In fact, even if the worst case is potentially intractable, our approach can be proficiently used in real-life scenarios because of the peculiar semantics of the policy.

Our future work aims to extend the model to other security contexts (e.g. VPNs) and to consider distributed scenarios too.

As a final note, many of the algorithms presented here are prone to parallelization (due to the properties of sets obtained as Cartesian products) and this can provide better performance on modern multi-core, multi-thread architectures.

Acknowledgements

Authors want to thank Alberto Cappadonia and Jacopo Silvestro for their excellent work in implementing application layer firewall functionalities and drafting an initial version of this paper. This work has been partially supported by the European Union project PoSecCo (EC contract IST-257129).

REFERENCES

[1] M. Ko and C. Dorantes, "The impact of information security breaches on financial performance of the breached firms: an empirical investigation," *J. of Information Technology Management*, vol. 17, pp. 13–22, 2006.
[2] Ponemon Institute, "Second annual cost of cyber crime study," Ponemon Institute Research Report, 2011.
[3] Center for Strategic and International Studies, "Securing Cyberspace for the 44th Presidency," 2008.
[4] E. Al-Shaer and H. Hamed, "Modeling and management of firewall policies," *IEEE Trans. on Network and Service Management*, vol. 1, no. 1, pp. 2–10, 2004.
[5] Check Point Software Technologies LTD., "SmartCenter." http://www.checkpoint.com/products/smartcenter/index.html
[6] C. Basile, A. Cappadonia, and A. Lioy, "Network-level access control policy analysis and transformation," *IEEE/ACM Trans. on Networking*, vol. 20, no. 4, pp. 985–998, 2012.
[7] R. Yavatkar, D. Pendarakis, and R. Guerin, "A Framework for Policy-based Admission Control," RFC-2753, 2000.
[8] M. G. Gouda and A. X. Liu, "A model of stateful firewalls and its properties," *IEEE Int. Conf. on Dependable Systems and Networks*, Yokohama (Japan), 2005.
[9] Squid Web Proxy Cache, "http://www.squid-cache.org/."
[10] A. Westerinen et al., "Terminology for Policy-Based Management," RFC-3198, 2001.
[11] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE JSAC*, vol. 21, no. 4, pp. 560–571, 2003.
[12] J. V. et al., "AAA Authorization Framework," RFC-2904, 2000.
[13] D. Wessels, *Squid: The Definitive Guide*. O'Reilly, 2004.
[14] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," RFC-1738, 1994.
[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2007.
[16] P. Linz, *An Introduction to Formal Languages and Automata*, 3rd ed. Jones and Bartlett Publishers, Inc., 2001.
[17] J. A. Brzozowski, "Derivatives of regular expressions," *J. of ACM*, vol. 11, pp. 481–494, 1964.
[18] S. C. Kleene, "Representation of Events in Nerve Nets and Finite Automata," *Shannon and McCarthy, Automata Studies*, pp. 2–13, 2006.
[19] A. Wool, "Trends in firewall configuration errors: Measuring the holes in swiss cheese," *IEEE Internet Computing*, vol. 14, pp. 58–65, 2010.
[20] D. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.
[21] A. M. ller, "Finite state automata and regular expressions for Java. http://www.brics.dk/automaton."
[22] J. D. Moffett and M. S. Sloman, "Policy Conflict Analysis in Distributed System Management," *J. of Organizational Computing*, vol. 4, no. 1, pp. 1–22, 1993.
[23] E. Lupu and M. Sloman, "Conflicts in policy-based distributed system management," *IEEE Trans. on Software Engineering*, vol. 25, no. 6, pp. 852–869, Nov. 1999.
[24] M. Sloman, "Policy driven management for distributed systems," *J. of Network and System Management*, vol. 2, no. 4, pp. 333–360, 1994.
[25] S. Hazelhurst, A. Attar, and R. Sinnappan, "Algorithms for improving the dependability of firewall and filter rule lists," *Int. Conf. on Dependable Systems and Networks*, New York (NY, USA), June 25–28, 2000, pp. 576–585.
[26] H. Adiseshu, S. Suri, and G. M. Parulkar, "Detecting and resolving packet filter conflicts," *INFOCOM*, 2000, pp. 1203–1212.
[27] F. Baboescu and G. Varghese, "Fast and scalable conflict detection for packet classifiers," *Comput. Netw.*, vol. 42, no. 6, pp. 717–735, 2003.
[28] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," *SIGCOMM'99*, Cambridge (MA, USA), September 1999, pp. 135–146.
[29] A. Mayer, A. Wool, and E. Ziskind, "Fang: A firewall analysis engine," *IEEE Symp. on Security and Privacy*, Oakland (CA, USA), May 14–17, 2000, pp. 177–187.
[30] A. Wool, "Architecting the Lumeta firewall analyzer," *USENIX Security Symposium*, Washington (DC, USA), August 13–17, 2001, 14 pages.
[31] A. Mayer, A. Wool, and E. Ziskind, "Offline firewall analysis," *Int. J. of Information Security*, vol. 5, no. 3, pp. 125–144, 2006.
[32] A. X. Liu and M. G. Gouda, "Firewall policy queries," *IEEE Trans. on Parallel and Distributed Systems*, vol. 20, pp. 766–777, 2009.
[33] A. R. Khakpour and A. X. Liu, "Quantifying and querying network reachability," *IEEE Int. Conf. on Distributed Computing Systems*, Genoa (Italy), 2010, pp. 817–826.
[34] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *IEEE JSAC*, vol. 23, no. 10, pp. 2069–2084, 2005.
[35] M. Benelbahri and A. Bouhoula, "Tuple based approach for anomalies detection within firewall filtering rules," *IEEE Symp. on Computers and Communications*, Aveiro (Portugal), July 1–4, 2007, pp. 63–70.
[36] S. Thanasegaran, Y. Yin, Y. Tateiwa, Y. Katayama, and N. Takahashi, "A topological approach to detect conflicts in firewall policies," *IEEE Int. Parallel & Distributed Processing Symp.*, Rome (Italy), May 23–29, 2009, 7 pages.
[37] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "Fireman: A toolkit for firewall modeling and analysis," *IEEE Symp. on Security and Privacy*, Oakland (CA, USA), May 21–24, 2006, pp. 199–213.
[38] M. G. Gouda and X.-Y. A. Liu, "Firewall design: Consistency, completeness, and compactness," *IEEE Int. Conf. on Distributed Computing Systems*, Tokyo (Japan), March 24–26, 2004, pp. 320–327.
[39] A. X. Liu and M. G. Gouda, "Complete redundancy detection in firewalls," *IFIP Data and Applications Security conf.*, Storrs (CT, USA), August 7–10, 2005, pp. 193–206.
[40] M. Abedin, S. Nessa, L. Khan, E. Al-Shaer, and M. Awad, "Analysis of firewall policy rules using traffic mining techniques," *Int. J. Internet Protoc. Technol.*, vol. 5, no. 1/2, pp. 3–22, 2010.
[41] J. G. Alfaro, N. Boulahia-Cuppens, and F. Cuppens, "Complete analysis of configuration rules to guarantee reliable network security policies," *Int. J. of Information Security*, vol. 7, no. 2, pp. 103–122, 2008.
[42] J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, and S. Preda, "Mirage: a management tool for the analysis and deployment of network security policies," *Int. workshop on data privacy management*, Leuven (Belgium), September 15–16, 2011, pp. 203–215.
[43] H. Hu, G.-J. Ahn, and K. Kulkarni, "Detecting and resolving firewall policy anomalies," *IEEE Trans. on Dependable and Secure Computing*, vol. 9, no. 3, pp. 318–331, 2012.
[44] A. K. Bandara, A. C. Kakas, E. C. Lupu, and A. Russo, "Using argumentation logic for firewall configuration management," *IFIP/IEEE Int. Symp. on Integrated Network Management*, Long Island (NY, USA), June 1–5, 2009, pp. 180–187.
[45] H. Hu, G.-J. Ahn, and K. Kulkarni, "Ontology-based policy anomaly management for autonomic computing," *Int. Conf. on Collaborative Computing*, Orlando (FL, USA), October 15–18, 2011, pp. 487–494.
[46] F. Cuppens, N. Cuppens-Boulahia, J. Garcia-Alfaro, T. Moataz, and X. Rimasson, "Handling stateful firewall anomalies," *IFIP Int. Information Security and Privacy Conf.*, Heraklion (Greece), June 4–6, 2012, pp. 174–186.
[47] L. Buttyán, G. Pék, and T. V. Thong, "Consistency verification of stateful firewalls is not harder than the stateless case," *Infocommunications Journal*, vol. LXIV, no. 2009/2-3, pp. 1–8, 2009.

Cataldo Basile holds a M.Sc. (*summa cum laude*) and a Ph.D. in Computer Engineering from Politecnico di Torino where is currently a research assistant. His research is concerned with policy-based management of security in networked environments, policy refinement, general models for detection, resolution and reconciliation of specification conflicts, and software security.

Antonio Lioy is full professor at Politecnico di Torino, where he leads the TORSEC research group active in information system security. His research interests include network security, public-key infrastructure (PKI), and policy-based system protection. Lioy received a MSc in Electronic Engineering (*summa cum laude*) and a PhD in Computer Engineering, both from Politecnico di Torino.