ForestFirewalls: Getting Firewall Configuration Right in Critical Networks

Paper #30, 12 pages

ABSTRACT

Firewall configuration is critical, yet often conducted manually with inevitable errors, leaving networks vulnerable to cyber attack [36]. The impact of misconfigured firewalls can be catastrophic in Supervisory Control and Data Acquisition (SCADA) networks. These networks control the distributed assets of industrial systems such as power generation and water distribution systems. Automation can make designing firewall configurations less tedious and their deployment more reliable. However, current research gaps prevent firewalls from being automatically configured.

In this paper, we propose *ForestFirewalls*, a high-level approach to configuring SCADA firewalls. Our goals are three-fold. We aim to: first, decouple implementation details from security policy design by abstracting the former; second, simplify policy design so as to encourage good design; and third, provide automated checks, pre and post-deployment, to guarantee configuration accuracy. ForestFirewalls meets these goals by automating the implementation of a policy to a network and by auto-validating each stage of the configuration process. We test our approach on a real network to demonstrate its effectiveness in simplifying and automating the configuration of SCADA firewalls.

1. INTRODUCTION

"The single most important factor of your firewall's security is how you configure it."

Rubin and Greer [32]

Supervisory Control and Data Acquisition (SCADA) networks control the distributed assets of many industrial systems. Power generation and water distribution are just two examples that illustrate the critical nature of these networks. Others include factory automation, sewage management, airport control systems and chemical plant control.

SCADA networks are not like corporate IT networks [34]. IT networks can accept a degree of reliability orders of magnitude lower than the network controlling a power station. A fault in the latter will cost serious money, if not lives.

At the same time, SCADA networks often incorporate highly vulnerable devices. The Programmable Logic Controllers (PLCs) that control physical devices such as gas valves have highly constrained memory and computational power. Today, they often include network functionality such as a TCP/IP stack, but cannot possibly contain sophisticated security functionality.

Despite their name, PLCs are not *user* programmable. The plant operator does not program them – that requires a programming board that pushes low-level code into an EPROM or the like. The devices come pre-installed with code (and security holes). There are many PLCs in a power station, along with similar devices providing telemetry, and their upgrade is likely to occur during a major overhaul of a plant, which might happen once in a decade (if that often).

We can see that these devices would be incredibly vulnerable if exposed to the wide world, and a plant operator cannot fix the vulnerabilities on the devices. Air gaps have been proposed as a solution to protect these devices, but an air gap is no longer a feasible approach for many reasons. In fact, Byres calls the idea a myth [8] to emphasise how poor a solution it is.

The only viable approach today is a firewall, or series of firewalls [9, 34].

As Rubin and Greer note [32], it is, therefore, vital that these firewalls are configured correctly. Misconfiguration of SCADA firewalls can lead to security breaches, resulting in significant environmental damage, financial loss or worse, the loss of human lives. An example past incident is the hacking of Maroochy Shire Council's sewage system in 2000, which saw tonnes of raw sewage released into public parklands and river systems [22]. Other examples include the sophisticated Stuxnet worm which attacked and damaged Iran's nuclear facilities in 2010 [34], and the hacking of a German steel mill in 2014 that destroyed its blast furnace [5].

Unfortunately, firewall configuration, in practice, is a complicated and repetitive manual task. It involves training in proprietary and device specific configuration languages, and long and complex device configurations. Lack of automation tools to assist this task has resulted in unoptimised, error-prone configurations [2, 36, 37].

The problem is exacerbated in SCADA plants where industrial engineers generally lack specialised networking and security knowledge. Such knowledge is often brought in through third party contractors. These IT security specialists, on the other hand, are not familiar with the particular requirements of industrial engineering, and are on-site only for brief periods.

A cost-effective alternative to training plant engineers to become IT specialists is to build network operations tools that derive firewall configurations from high-level policy. High-level configuration approaches using SDN have been proposed [25,33], but they remain a distant reality for SCADA networks, where TCP is a recent innovation. And power plants are insecure now [2]! SCADA networks need a solution that works now, using off-the-shelf technology. In this paper, we propose such a solution: *ForestFirewalls*.

Our system provides a mechanism for specification of security policy at a level any non-IT specialist could understand. What's more, it forces good designs on its users through principles derived from the study of real SCADA firewall configurations [2] and International Society for Automation (ISA) best practices [3,7,34]. Most notably:

- *Single source of truth:* more specifically, "Security managers need a single place to look for the corporate policies on who gets in and who doesn't." [20]. This is a general principle in computer science [6], and it applies doubly so here.
- *Simplify:* we don't try to provide every possible security feature or knob. At best, advanced features create confusion, and at worst, bad implementations can create security flaws.
- *Verify everything again and again:* there is a clear danger in assuming any one piece of software functions correctly, from the firewall up to and including our own system. We check the configuration works at every level possible.
- *No implicit rules:* implicit rules allow unexpected interactions, and undesirable consequences [2]. Desired flows must be explicitly allowed.
- *Rule order should not matter:* it should be possible to add, or subtract a policy rule without considering its effect on every other rule! Surprisingly, none of the existing firewall configuration platforms we checked [11,13,14,23] achieved this. Operators using these tools, have to provide correct rules, and also maintain correct rule order, to avoid unexpected interactions.
- *Separate structure from function [30]:* decoupling 'structure' (*i.e.,* network topology) from 'function' (*i.e.,* policy specification) allows policies to be specified in high-level requirements, instead of network-centric minutiae like IP addresses.
- *Convenience:* security and convenience are usually at odds, but wherever possible convenience should be provided. This is not a luxury – lack of convenience is one of the main reasons operators circumvent their own security, creating flaws in that security.

Our system comprises a suite of tools to write policy, validate, test configurations, and create real configurations. It already provides some of the core functionalities required in this domain, but is extensible.

We demonstrate it with a real example, derived from the actual (but anonymised) firewall configurations of a real SCADA plant. The example is intentionally small for clarity, but it shows a core set of functionality. The example includes several zones, two firewalls, and multiple real services much as they would run in the real network. Our testbed uses two different firewalls: one Cisco and one Linux-based, in order to show both the device independent nature of our policy language, and that heterogeneous network devices can be configured in the same network. The network offers multiple services: routing, DNS, HTTP, HTTPS, FTP, Oracle, *etc.,* and we use test traffic (both allowed and disallowed) on the network to show correct function.

The proof of the pudding is that we can specify all of the policy for this network in only 68 Lines of Code (LoC), to generate the equivalent of 2720 device-level LoC found in a real SCADA case study [2]. This order of magnitude reduction, along with rigorous validation, shows the value of *ForestFirewalls*.

2. RELATED WORK

Firewall vendors have introduced many products and security management tools with varying levels of sophistication [11, 13, 14, 23]. But, security policies still cannot be specified flexibly enough and in detail using high-level requirements through these tools.

Several attempts have been made at high-level configuration of firewalls. For one, Cisco introduced security levels for quick and easy access between internal and external firewall interfaces [13], but these cannot specify detailed traffic restrictions. Hence, Access Control Lists (ACLs) are required to supplement these levels. Security levels may also not map to clear security policies. This hinders firewall autoconfiguration, which needs clear policies [6] to admit traffic.

The problem of firewall configuration is well studied. Fang [29] and Lumeta [35] are interactive management and analysis tools that run queries on firewall rules. They accept a network topology description and firewall configurations to detect firewall errors, but do not address the root cause: the manual and device-centric configuration approach.

Tesseract implements a network control plane that enables direct control of Ethernet and IP based services [38]. It promotes centralised policy implementation, but lacks the ability to abstract low-level policy configuration details.

SANE uses a central Domain Controller with trusted privileges [10] to reduce end-host initiated attacks in corporate networks. It supports topology-independent high-level declarative policies, but provides no assurance of expected configuration behaviour prior to deployment. Such assurance, even via simple automated emulations, is essential in SCADA networks where downtimes must be minimised.

Firmato [4] employs a network grouping language that is independent of the firewalls and routers used in the network. However, the specification also relies on minute details such as IP addresses as input.

The network programming language introduced in the Frenetic project [16], is able to capture dynamic policies, but cannot assist with queries related to network reachability. NetKAT is a language [1] built on a complete equational theory to address this shortfall. It additionally supports features such as traffic isolation and compiler correctness. But the language is not specifically aimed at configuring firewalls and does not provide means to generate filtering rules.

Network virtualisation requires traditional network boundaries to be broken, to allow N:1 mapping between operating systems (*i.e.,* VMs) and a network port. Cisco has introduced security policy management products (*e.g.,* VNMC for VSG policy management) to cater for the complexity this introduces to network management [15]. For scalability, the products allow VMs to be allocated to zones and policies to be defined per zone. However, each VM still needs to be defined using low-level detail such as hostnames.

Most related works do not propose a high-level description that intuitively decouples policy from topology. In some cases [4], topology needs to be explicitly mapped to policy through the specification per host/subnet basis. There is also no automated pre- and post-deployment verification of policies. Moreover, none of the above works examine SCADA networks, with unique security requirements and best practices compared to Corporate networks.

The prior work does not address some of the practical issues. Most notably, complexity. Firewall vendors have concentrated on new and impressive features to create systems with as much or more complexity as the base firewall configurations.

Our research aims to tackle the problem head-on. The solution we propose, *ForestFirewalls*, uses security abstractions to drastically reduce firewall policy complexity. It supports a vendor and device independent policy specification platform that is easy to use, yet powerful. Our system also supports automated verification to reduce firewall misconfigurations. We aim to make firewall configuration a commodity skill rather than a specialisation, so business managers and plant engineers alike can manage their SCADA firewalls.

3. REQUIREMENTS

Bush and Bellovin [6] investigated the core requirements of an automated security configuration system. They identified the following:

- *Clear policies:* an automated system cannot resolve between a plausible and a correct policy [6]. For example, between allowing HTTP access to a publicly shared Web server or to a sensitive internal Web server. So the policy must be clearly understood by a Manager.
- *Database driven:* all device configurations and their changes must be recorded in a database [6]. Creating a *single reference point* for configuration data encourages fast response to security incidents as well as accurate security audits.
- *Meta-configurations:* specifications or instructions about real configurations need to be obtained by abstraction and parameterisation.

However, there are extra issues not described by [6]. For one, there is an assumption that the auto-configuration system generating the configurations is correct. Correctness must stem from configuration validation, as we describe next.

3.1 Policy verification

SCADA operators need assurance that the device configurations generated produce the expected security outcome, both pre- and post-deployment. Multiple verification stages (Figure 1) can provide this assurance.

Upper Verification Tier (SCADA best practices, Alloy)
Middle Verification Tier (Netkit emulations, pathological-traffic tests)
Lower Verification Tier (real network, live-traffic tests)

Figure 1: *Policy verification tiers.*

Upper verification tier: it is important initially to check a specified SCADA firewall policy against available industry best practices [9,34]. Direct violations of best practices indicate exploitable vulnerabilities of the network implementing the policy, and should be prevented. Best practice violations can be accurately identified by conducting equivalence and inclusion checks on the canonicalised policies. See §7.

Complex firewall policies also produce unintended consequences through rule overlaps [27,39]. So it is additionally necessary to check policies from high-level through firewall-level for inconsistencies. We do so accurately, using a mathematical and logic based formal tool: *Alloy* [21].

Middle verification tier: the second stage help debug configuration problems prior to deployment. Network emulation offers a cost effective method to test configurations before actual deployment [24]. The *Netkit* open source software package [31] provides such an emulation platform with virtual devices and interconnections via User Mode Linux (UML). Automated pathological traffic tests, together with Netkit emulations, can verify that the generated configurations produce the expected outcome prior to deployment.

Lower verification tier: the final stage guarantees that the real firewalls operate as intended, post deployment. The automated tests are extended from emulations to the real network, to generate live-traffic and reveal unexpected configuration behaviour in the real firewalls.

Automated verification can drastically reduce the number of firewall misconfigurations. It can be used to identify best-practice violations and adverse policy interactions, and prevent those from propagating through to the firewalls. Most importantly, it provides users with a guarantee of the security outcome prior to implementation. This is particularly useful in successfully responding to intrusions and attacks.

A second issue not described by [6], but key to autoconfiguration, is the need to decouple policy from the network implementation. We discuss this in detail next.

3.2 Decouple policy from network

In practice, network architects and business managers decide what type of services are allowed through firewalls. Network engineers then implement these policies in the SCADA network. Intuitively, separation of the network intricacies from policy specification better suits these distinct phases. Conceptually this is analogous to the separation of architects and building contractors in construction. Contractors don't usually decide what roof shape a building should have!

Network topology can change often in response to new business needs, upgrades and service demands. This may alter the devices, services and administratively assigned parameters such as IP addresses and hostnames in the network.

Comparatively, security policies are static, changing mostly to meet business imperatives. These policies commonly only involve dozens of distinct services [2], making policy complexity relatively low compared to that of the network. This relative simplicity and invariant nature leads to decoupling policy from the network. The adage *"Structure and function should be independent"* [30] truly applies here.

Decoupling *structure* from *function*, has these advantages:

- *Policy specifiable via high-level, vendor neutral*[1] *requirements:* assists management-level policy makers not fluent in network-centric details.

[1]Vendor platform and device independent requirements.

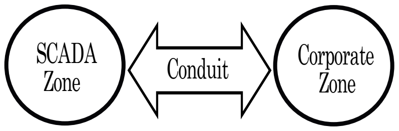


Figure 2: *A Zone-Conduit model, adapted from [7].*

- *Centralises policy management and promotes reuse:* a single topology-independent policy (*i.e., source of truth*) can be maintained for an organisation and applied across sites.
- *Streamlines network changes and upgrades:* the policy can be quickly re-mapped to a new topology, retaining previous levels of protection.
- *Simplifies best-practice enforcement:* best practice standards can be *precisely* specified in absence of proprietary details of specific networks.

Security abstractions are key to decoupling policy from network as we discuss in the next section.

4. SECURITY ABSTRACTIONS

A good high-level security abstraction captures the underlying network security concepts naturally and concisely. For one, in real networks, we might group systems with a similar set of traffic services enabled. For another, traffic restrictions between two systems may be enforced by a single or a series of firewalls. A good abstraction should decouple *what* is restricted between systems from *how* it is restricted.

The American National Standards Institute (ANSI)/ ISA standards introduce the *Zone-Conduit abstraction* as a way of segmenting and isolating the various sub-systems in a control system [3]. We conducted real SCADA firewall configuration case studies [2] using the Zone-Conduit abstraction, to evaluate its suitability for auto-configuration. A key finding was that the ISA Zone-Conduit model in its original specification is too flexible for automation. To increase its precision, we needed to add several extensions [2].

We describe our modified version here.

4.1 The ISA Zone-Conduit model

A *zone* is a logical or physical grouping of an organisation's systems with similar security requirements, based on criticality and consequence [3]. By grouping systems in this manner, a *single zone-policy* can be defined for all members of a zone. For example, 3 disjoint security zones can be defined to accommodate low, medium and high-risk systems, with each device assigned to its respective zone, based on their security level needed. A low-risk system can be accommodated within a medium or high security zone without compromising security, but not vice versa.

A *conduit* provides the secure communication path between two zones, enforcing the policy between them [3]. Security mitigation mechanisms (*e.g.,* firewalls) are implemented within a conduit. A conduit could consist of multiple links and firewalls, but logically is a single connector. Conduits abstract *how* a policy is enforced, so we can focus on *what* needs to be enforced.

Figure 2 shows two typical zones in a SCADA network, the SCADA-Zone and the Corporate-Zone, linked by a conduit.

4.2 Our modifications to the model

Through real SCADA firewall configuration case studies [2] we found that ISA Zone-Conduit model in its original specification is too flexible for automation. For one, the ISA model allows alternate ways of defining zones and conduits to cater for business models. It loosely permits 1:n or n:1 mapping between conduits, firewalls and policy.

To refine the model, we introduce several extensions [2]. First, we need to enforce a 1:1 mapping between policies and conduits. Second, dedicated Firewall-Zones are required to capture firewall management policies. Third, Abstract-Zones are required to capture the distinct policy requirements of serial firewalls. Carrier-Zones are also necessary to abstract any carrier based transit outside of an administrative domain's control. With these revisions, the best practice produces a tight specification suitable for auto-configuration.

4.3 Single zone-policy

Our approach is to strictly enforce a *single zone-policy*. This implies that selected subsystems in a zone (*e.g.,* a server) should not have their own separate policies (*i.e.,* no exceptions). Allowing exceptions would impart a false sense of security to those systems. These systems are only as secure as the zone itself, in the absence of any firewalls enforcing a real separation.

A *single zone-policy* leads to every device within a zone to have same set of permissions to initiate, accept or block one or more services. This property allows us to specify policies simply and unambiguously using *inter-zone flows*.

4.4 Positive, explicit policies

We can further restrict *inter-flows* to express positive abilities[2] and all flows not explicitly allowed are *explicitly denied*.

Policy specification is much simplified by use of positive, explicit flows. These flows render the rule order irrelevant in a policy. A policy now holds the same semantics, irrespective of how its rules are organised. Hence, policy makers need not concern about the order when adding or removing policy rules. By being explicit, we also guard against services being accidentally enabled implicitly or by default.

4.5 Isolation of traffic

Inter-zone flows can be grouped by *(source-zone,dest-zone)* tuples. The policy of each tuple reflects a piece of the network that can be programmed independently from the rest of the network, representing a high-level *network-slice* [19]. In the example policy below, `(Z1,Z2)` and `(Z2,Z3)` represent network slices.

```
Policy A {  Z1 -> Z2 : https, dns;
            Z2 -> Z3 : http, ftp, dns; }
```

Network-slices allow modular construction of policies that help deliver guarantee of traffic isolation for specified zones. Traffic isolation restricts the set of destinations that a packet may reach. For instance, a packet applicable to a slice is only processed by the policy of that slice. In the above example, traffic flow from `Z1` to `Z2` is controlled by the policy of slice `(Z1,Z2)` only. Therefore, to identify what destinations packets are allowed to reach from `Z1`, we simply need to consider such slices containing `Z1` as the source zone.

Firewall policy complexity is reduced by an order of magnitude by the use of our precision-increased Zone-Conduit model. This model is easy to understand, maps cleanly to

[2]Refers to the ability to initiate or accept a traffic service.

topology and abstracts network and vendor specific intricacies well. In contrast to the traditional configuration approach, the model allows users to focus on *what* policy to specify without concerning with *which firewall* to implement it on or *how* to implement it.

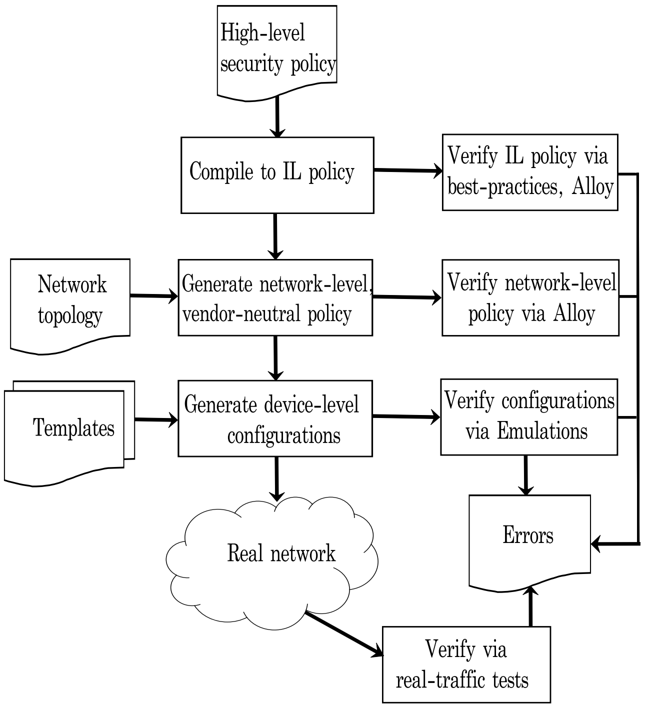
5. SYSTEM OVERVIEW

We now describe our auto-configuration system design as depicted in Figure 3, with the details outlined below:

High-level security policy: The topology independent policy input file created using our high-level specification. Policy details are discussed in §6.

Compile to intermediate-level (IL) policy: Parses the high-level policy to an intermediate format that can be checked.

Network topology: The input network topology described in the XML-based graph file format *GraphML* [18]. The file contains information of all devices of the underlying network and their interconnections. The crucial aspects are the details of the topology near the policy enforcing firewalls.



High-level security policy
Compile to IL policy
Verify IL policy via best-practices, Alloy
Network topology
Generate network-level, vendor-neutral policy
Verify network-level policy via Alloy
Templates
Generate device-level configurations
Verify configurations via Emulations
Real network
Errors
Verify via real-traffic tests

Figure 3: *Firewall auto-configuration process.*

Generate network-level, vendor-neutral policy: Translates high-level policy to network-level, by coupling policy to the input network topology. See §5.1.

Verify IL policy via best-practices, via Alloy: Formally checks an IL policy for SCADA best-practice violations and for correctness. Best-practice checks employ canonicalised policies. Policy correctness is validated using a a mathematical and logic based tool: Alloy [21]. Alloy can reliably find anomalies within a policy. See §5.2.

Verify network-level policy via Alloy: Formally checks network-level policy for correctness via Alloy [21]. See §5.2.

Device templates: A repository of meta-configurations for various vendor and device platforms. Cisco ASA5505 and UML IPTables models are currently supported, but the system is easily extensible.

Generate device-level configurations: The rendering of device-specific configurations for firewalls using the network-level policy and the device templates.

Verify via emulations: Device configurations are pushed to an emulated network for pre-deployment testing. Test scripts are auto-executed in this network, to generate pathological traffic and validate configurations. See §5.3.

Real network: Device-specific configurations are pushed to hardware in a real network. At present this is conducted manually[3], but we intend to automate it.

Verify via real traffic tests: Automated tests are created for the real-network, generating real-traffic, to verify post-deployment behaviour of firewall configurations. See §5.3.

5.1 Network-level policy generation

A high-level policy is implemented on a network by coupling the security policy to the network topology instance. The resultant network-level Access Control List (ACL) rules are vendor/device neutral. A generic format allows easy checking of rules for inconsistencies. The policy generation steps are outlined below.

5.1.1 Zone-Conduit model construction

The first step is to generate the Zone-Conduit model of the input network. To do so, we temporarily ignore the firewalls and their connecting links in the topology. This leaves a collection of connected components, each reflecting a disjoint security zone in the network. Once all disjoint zones are identified, we build a preliminary *Zone-Firewall model*, containing these zones and their firewall interconnections [2]. Additional Firewall-Zones, Abstract-Zones and Carrier-Zones are added to the model as required.

The conduits in the network are then defined. A conduit is not necessarily an atomic device, but it implements a single security policy between two zones.

There is no guarantee that the Zone-Conduit model generated for the input network will *always* match that perceived by the policy creator. Hence, we must cross check the real model against that provided through the specification. If mismatched, an error is reported indicating incompatibility.

5.1.2 Network coupling and rule translation

As the zones and conduits are identified in the input network, an implicit mapping is created between each zone and its host/subnet composition. This mapping readily translates the high-level policy to the underlying network. The source and destination zone of each high-level rule can be replaced with their corresponding IP address ranges from this mapping. Then by taking the cross product of these IP address ranges, with the original rule operator and service description, the equivalent network ACL rules are generated.

Multicast rules may also be required for the correct operation of certain protocols. For instance, when OSPF is specified as a dynamic routing protocol by the user, multicast rules are required to enable neighbour relationships to correctly form within a single OSPF area. Similarly, stateful protocols (*e.g.,* TCP) require return path rules in addition

[3]Automation of pushing device configurations is more development than research.

to the forward path rules for correct operation. *ForestFirewalls* handles these requirements automatically, generating and incorporating any supplementary rules as necessary.

5.1.3 Path selection and conduit configuration

The system identifies possible communication paths in the Zone-Conduit model per high-level policy rule. Paths that are deemed impractical are eliminated. For instance, (i) traffic cannot transit a Firewall-Zone. Firewall-Zones only enable traffic flow to and from the firewall but cannot forward traffic, (ii) a traffic path cannot form loops around firewalls. If a path requires a traffic packet to traverse a particular firewall interface more than once, it is discarded, (iii) traffic originating from or terminating at a Firewall-Zone must have a valid external path through the network.

Using valid paths, the system configures all conduits. By default, each conduit implements a *deny all* policy between its interconnecting zones, relying on explicit flows to enable traffic. The strategy creates a *defence in depth* security architecture [3, 9], preventing a single point of failure that can trigger cascading security breaches across the network.

We evaluate the firewall interface layout within each conduit to determine how ACL rules need to be placed (inbound or outbound) on the respective firewall interfaces. This ability to configure a group of firewalls at once, makes *ForestFirewalls* scale at lower cost.

Our high-level policy is easily adapted to incorporate new zone additions to a network. The updated policy is fast re-mapped to the network to protect the new zones.

5.2 Formal policy verification

Policy rules can also give rise to unintended consequences through rule overlaps. Overlaps can be classified as *redundancies* or *conflicts* [27, 39]. Redundant rules can be removed without affecting the semantics of a policy. Such rules reflect configuration inefficiencies and cause potential confusion in policy specification. A *conflict* occurs when a rule overlaps with preceding rules but specifies a different action, creating ambiguity.

Our system only supports *positive permissions*. So, conflicting rules are not an issue. Correct ordering of rules is largely required to avoid rule conflicts. We remove conflicts by design, rendering rule order irrelevant. Redundancies are still possible so a policy needs to be checked for these.

A policy needs to be checked against SCADA best practices for compliance. We do so accurately, by first deriving the canonical forms of the input and best-practice policies. Then we conduct inclusion checks and equivalence checks to identify any violations. For details, see §7.

A policy needs to be verified for correctness, both at a high-level and a network-level. At a high-level, the rules specified using flows may contain inconsistencies. At a network-level, the generated ACLs rules can include redundancies. We employ a model checker: Alloy [21], to do our testing.

Formal model-checking is generally complex, so Alloy attempts to find counter-examples to illustrate problems. Essentially it's a refuter [21] not a prover. But, its ability to comprehensively analyse a model, even within finite bounds, makes it very useful in both research and academia [21]. For Alloy based verification, see §7.

5.3 Pre- and post-deployment testing

Pre-deployment testing of generated configurations requires test-candidates to be selected from the input network to suit the specified policy. We consider the source and destination zone of each high-level rule and select devices. For example, in the policy below, there must be one or more devices in `SCADA_Zone` that are capable of initiating HTTP traffic (*i.e.,* HTTP test-clients) and others in `Corporate_Zone` that are capable of accepting HTTP traffic (*i.e.,* HTTP test-servers).

```
SCADA_Zone -> Corporate_Zone : http
```

Once selected, test candidates are auto-configured to generate/accept pathological traffic. For instance, we use *Proftpd* for a FTP test-server and configure it with appropriate FTP control and data ports. Its corresponding FTP-client has *lftp* configured with the required FTP login details.

Pre-deployment tests are conducted using an emulated network. Netkit is an open source network emulator that enables virtual devices and interconnections using UML [31]. AutoNetkit is a tool designed to automate emulated network experimentation via Netkit [24]. Our system uses an extended version of AutoNetkit, with basic firewall capabilities, to generate emulations.

Once the emulated network is running, automated tests verify expected firewall configuration behaviour. *ForestFirewalls* uses *Expect*: a UNIX scripting and testing utility, to generate these test-scripts. Expect enables automated interactions with programs that expose a text terminal interface [26]. Netkit launches these test scripts within a Netkit Virtual Machine (VM), once the VM is running. The scripts run sequentially, with independent outcomes (*i.e.,* failure of one does not affect another, so their ordering is irrelevant).

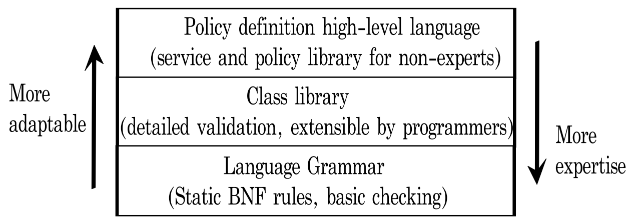
Expect test scripts verify that the permits rules in a policy works correctly (*i.e.,* positive vetting), but we still need to check that all other services not explicitly enabled are blocked. This negative vetting is conducted using automated, exhaustive port-scans employing *nmap* and *tshark*.

The same test-suite can be used in the real network, post configuration deployment. The tests now generate live-traffic, verifying expected real-firewall behaviour.

6. POLICY SPECIFICATION FRAMEWORK

A useful network policy specification framework should cater for management-level policy makers as well as competent programmers. Policy makers need to define high-level policies to meet business goals. Programmers may wish to extend the framework to add more features. A layered approach (Figure 4) supports both cases.

6.1 A layered approach



Policy definition high-level language
(service and policy library for non-experts)
Class library
(detailed validation, extensible by programmers)
Language Grammar
(Static BNF rules, basic checking)
More adaptable
More expertise

Figure 4: *Policy specification in layers.*

Policy definition high-level language: designed primarily for non-expert users to define services and security policies, it uses a library of services and security policies in conjunction with a very simple language. The service library consists of Internet Assigned Numbers Authority (IANA) well-known services and the policy library contains common SCADA security policies, all easily extensible by a non-expert user. The language syntax and semantics are also intuitively simple for non-expert users. Informative warnings and errors are returned for fast debugging. See §6.2.

Class library layer: dedicated to expert Programmers, this layer features an Object Oriented Programming (OOP) based, well-defined object hierarchy that consists of rules for constructing protocols (*e.g.,* TCP, UDP) and services. Detailed checking of object specific attributes (*e.g.,* TCP/UDP port numbers are between 0-65535) are handled by their respective classes. A direct mapping between the grammar rules and the Classes makes the library easily extensible, but it is only intended that expert protocol engineers would extend this. Most operators would use the higher layer.

High-level language grammar: dedicated to the language designers, this layer consists of Backus-Naur Form (BNF) rules that control the language semantics. The grammar includes basic checking (*e.g.,* argument length, null checks), but delegates detailed checking to the class library layer. The rules are static and can only be altered by the language designers as needed. This preserves the original objectives of a high-level specification which is intended to change slowly.

Our layered policy-specification architecture leads to a vendor and device neutral policy-specification framework. The system suits naive users, but the framework is easily extensible to cater for new network applications and protocols.

6.2 ForestFirewalls high-level language

Simply put, the *ForestFirewalls* specification language allows a user to instantiate a high-level security policy. Below is the definition (a complete example can be found in §8).

ForestFirewalls' parser is currently implemented in Python and Ply (a Python specific implementation of `lex` and `yacc`). It translates a *ForestFirewalls* specification (*i.e.,* a `.policyml` file) into its Intermediate Language (IL) representation using object definitions from the underlying class library, also implemented in Python.

6.2.1 *Service and Service-group description*

A service is defined using:

```
service <service-name> { protocol=<protocol-base>;
                       <protocol-attributes-list>; }
```

For example, a custom implementation of HTTP, based on the above service description format is given by

```
service custom_http {  protocol=tcp;
                       tcp.dest_port=8080;
     comment=‘‘Internal Web service’’; }
```

All unspecified attribute values have defaults assigned (*e.g.,* here `tcp.source_port=0-65535`). Service specific comments are enabled via the `comment` field. This type of code documentation allows commentary in the lower tiers to be auto-generated. The aim is to help document network and device level firewall rules to avoid the common problem that rules cannot be deleted because no one remembers why they exist.

ForestFirewalls prohibits the description of *generic services* such as *all-TCP* or *all-IP* for several reasons. For one, SCADA case studies [2] reveal that users exploit generic rules where possible for convenience, such as allowing *all-IP* traffic just to enable EIGRP traffic. Far more services than necessary are thus admitted through firewalls to achieve a simple end goal, creating security vulnerabilities.

Secondly, such inherently broad services don't contribute towards forming well-defined security policies. They cloud the ability to accurately see the type of cyber threats a network is being protected from.

Services can also be grouped using a service-group:

```
service_group <group-name> {
 <service-name1>, <service-group-name1>, ... }
```

Service-groups provide a level of indirection, so that precise application protocols used to achieve network functionality (*e.g.,* file transfer) can change without needing policy alterations. They also provide convenience but are explicit.

A service-group is represented as a set allowing set operations: union (`,`), intersection (`^`) and difference (`\`). A new service group can be created by applying these operators on already defined service groups.

The following snippet defines a service-group containing various file transfer services:

```
service_group file_transfer { ftp, http, smb }
```

6.2.2 *ForestFirewalls library file imports*

Non-expert programmers may not be familiar using a complex namespace library with many features. We developed a namespace hierarchy (partially shown in Listing 1) that is simple enough for them, yet provides rich features for managing and reusing namespaces.

The `system` namespace contains generic library definitions for all users and is located inside the global namespace `forestlib`. It has a fully qualified name of `system`. An immediate member of `system` is `services`. Hence, it has a fully qualified name of `system.services`. Shown in Listing 1, `iana_services` is a member of `system.services` and has a fully qualified name of `system.services.iana_services`.

```
forestlib (global)
| system
| | services
| | | iana_services
| | | | http
| | service_groups
| | policy_rules
| custom_namespace1
```

Listing 1: *ForestFirewalls namespace hierarchy.*

ForestFirewalls assigns a default namespace for each custom specification created by a user. This namespace is based on the specification file name and it's placed directly under the global namespace (*i.e.,* at the same level as `system`). This flat design structure automatically prevents namespace duplication as two files with the same name and extension cannot be located at the same level of the global directory.

Importation facilitates reuse of policy. For instance, we can imagine the ISA creating a best practice ruleset for SCADA as a baseline for new installations.

6.2.3 *Zone-group description*

A zone-group groups a set of zones or other zone-groups (*i.e.,* nesting allowed) and is defined using

```
zone_group <group-name> {<zone-or-group-list>}
```

Nested zone-groups are iteratively resolved to a set of disjoint zones. Multiple zone-group declarations are checked for duplicate functionality to minimise code redundancy.

The snippet below describes an example zone-group, depicting `three_zones`: a set of zones in a network which is made up of 3 disjoint zones.

```
zone_group three_zones { corp_zone, scada_zone, dmz }
```

We also allow similar syntax and set operators for defining groups of TCP/UDP ports or ICMP types.

6.2.4 Policy-rule description

A high-level policy rule can be defined as below, where `operator` indicates the direction explicitly. `service-or-group-name` and `zone-or-group-name` shown, enable service(s) between the specified zones as per the `operator`.

```
policy_rule <rule-name> {
     <first-zone-or-group-name> operator
     <second-zone-or-group-name> :
     <service-or-group-name>  }
operator == '->' || '<->'
     (allow-to, allow-to-from)
```

The following policy rule models the capabilities of a `Corp_orate_zone` with regards to `Web` traffic:

```
policy_rule corp_web_rule { Corp_zone -> DMZ : Web }
```

6.2.5 Policy description

A policy is used to hold one or more policy rules and can be defined using the following format:

```
policy <policy-name> { <rule1>, <rule2>, ... }
```

7. VERIFICATION

Once a high-level policy is parsed by *ForestFirewalls*, it is stored in IL code and needs to be checked for SCADA best-practice violations as well as for correctness.

7.1 Best-practice compliance

We need to compare the semantics of an input policy with that of the SCADA best-practice policy, to check for best-practice violations. Two policies can contain different rule sets, but have the same semantics underneath. For instance, consider Figure 5(a) which depicts an example service plot for a TCP based policy for the tuple- (source-zone=Z1,dest-zone=Z2, protocol=TCP). The policy consists of two overlapping services that create the overlapping rectangles shown. Together, these rectangles form a single (disjoint) rectilinear-polygon. We could tediously compare each valid point between plots, to compare such policies, but that would be highly inefficient. A more efficient approach would be to derive a unique representation of each service plot (*i.e.,* policy) and then make comparisons.

We derive a canonical representation of the policies by dissecting the polygons into partitions. Our aim here is to find a unique partition quickly rather than a guaranteed minimal partition. In policy terms, partitioning yields a set of non-overlapping services with clear semantics.

We dissect the polygon formed in our example policy (Figure 5(a)), into horizontal partitions as shown in Figure 5(b), using a Rectilinear-Polygon to Rectangle conversion algorithm [17]. Each horizontal partition is chosen to guarantee its uniqueness (provable by contradiction). Canonical policy elements are derived by translating each partition back to policy level.

Intra-policy verification can be done by conducting equivalence checks on the canonical policy components:

$$Canonical(Z1 \rightarrow Z2) \equiv Canonical(SCADA \rightarrow Corp)$$

We can also evaluate whether the input policy adheres to the SCADA best practice policy using the inclusion check:

$$Canonical(InputPolicy) \subseteq Canonical(BestPracticePolicy)$$

7.2 Policy correctness

We also generate an Alloy language specification (*i.e.,* `.als`) file for the IL policy. A partial snippet of this export is shown in Listing 2. It depicts a formal model with 3 signatures: `Service`, `PolicyRule` and `SecurityPolicy`. In our initial model, a `Service` has the basic members: `ip_protocol`, `source_port`, `dest_port` and `icmp_type`. Of these members, only `ip_protocol` is mandatory (the multiplicity keyword `some` requires at least one element).

A `PolicyRule` has 4 members: `zone1` and `zone2` to capture the zone names, an `operator` and a single `service` element. The `operator` is a set of integers containing $\{1\}$ (representing a uni-directional permit: `->`) or $\{1, 2\}$ (representing a bi-directional permit: `<->`). The bi-directional permit is deconstructed as `->` $\{1\}$ and `<-` $\{2\}$ for simplicity, but the latter operator is omitted in the high-level syntax.

The global constraints are partially shown (Listing 2, lines 17–27), stating that the universal set ($Univ$) of `PolicyRule` must be made up entirely of rules in the policy. Additionally, $Univ$ of `Service` must be made up entirely of services defined within `PolicyRule` objects.

Predicates can determine if two given rules or services overlap (not shown). `Service` overlaps are found by computing their intersection and testing if the result has members. *String* type members (*e.g.,* `zone1`, `zone2`) can be directly compared. `PolicyRule` overlaps are checked similarly.

A 'no_rule_overlaps' assertion (also not shown) is defined to check whether there are distinct rules with overlapping criteria. If found, the check returns a counter-example, indicating potential inconsistencies in the high-level policy. Counter-examples can be inspected through Alloy's user interface to identify the underlying cause(s).

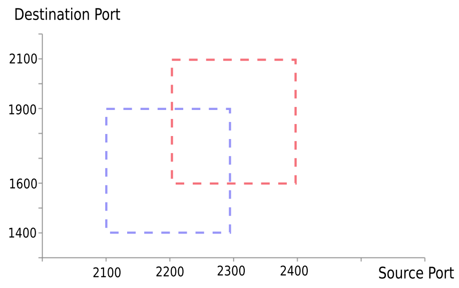
Currently, Alloy is itself run manually and output counter-examples help debug data. We do not auto-correct rules as this requires managerial discretion.

When overlaps are absent in the high-level policy, there should also be none in network-level policy, in theory. But we cannot simply 'trust' our system to always correctly generate network-level policy. So, we re-check the generated policy for overlaps and verify the fact.

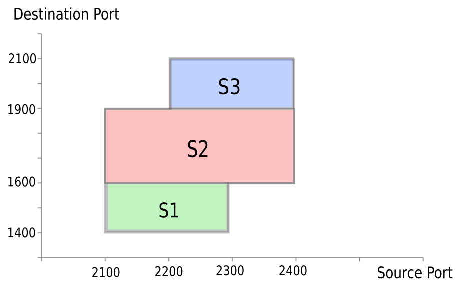
The Alloy export generated for network policy verification is similar to the high-level export. The key exception is the source and destination zone names are now replaced with IP address ranges in an `ACLRule`. Additionally, the `Service` signature also has members depicting protocol state. We also define an assertion here to check for `ACLRule` overlaps.

8. A CONCRETE EXAMPLE

We show here a concrete example, illustrating our methodology and the prototype system. The example is based on an actual SCADA case study [2] with the multi-firewall network configuration shown in Figure 6. Due to security con-



(a) TCP service overlaps form a polygon.



(b) Horizontal partitions of polygon in (a).

Figure 5: *Example canonicalisation of a TCP based policy for tuple- (source=Zone1,dest=Zone2,protocol=TCP).*

```
abstract sig Service {
   ip_protocol: some Int,
   source_port: set String,
   dest_port: set String,
   icmp_type: set Int
}
abstract sig PolicyRule {
   zone1: one String,
   zone2: one String,
   operator: some Int,
   service: one Service
}
// Policy definition
one sig SecurityPolicy { rules: some PolicyRule }

// List of global constraints
fact {

  // All defined rules are in the policy to check
  all r: PolicyRule | r in SecurityPolicy.rules

  // Policy rules make up universe of PolicyRule
  SecurityPolicy.rules = PolicyRule

  // A service belongs to at least one PolicyRule
  all s: Service | some r: PolicyRule | s in r.
       service
}
```

Listing 2: *High-level policy verification framework using Alloy (partially shown).*

cerns and non-disclosure agreements, a modified version of the real network is presented for discussion. Steps have been taken to keep the core security strategies identified intact. However, details such as IP addresses are anonymised.

R1 is a Cisco ASA 5505 firewall with 2 active physical interfaces pointing to `Corp` and `LAN`. R2 and GW are Linux IPtables firewalls with 2 active physical interfaces each. R2 points to `SCADA` and `LAN` while GW points to `Corp` and the Internet. The subnet summary is below.

The Corporate network (Corp): Provides access to business applications and the Internet.

Local Area network (LAN): Responsible for enabling connectivity between R1 and R2. The distinct vendor firewalls provide *defence in depth* [9] through multiple nodes of failure and firewall-software redundancy.

The SCADA network (SCADA): Responsible for providing networked access to plant equipment.

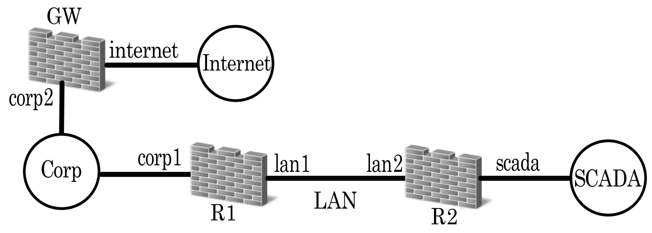


Figure 6: *The SCADA network under study.* ***`Corp`*** *and* ***`SCADA`*** *are the corporate and SCADA subnets while the firewalls are R1, R2 and GW.*

`Corp` and `SCADA` could accommodate up to 2,046 and 65,534 hosts respectively. `Corp` contains multiple management workstations, an HTTP server, an HTTPS server, an FTP server, an Email server, a syslog server and a DNS server. `SCADA` includes 2 `Oracle` database servers, management workstations and an HTTPS server.

8.1 Policy goals

The policy we consider is simple, but nonetheless covers many of the aspects that occur in more complex, real-life SCADA policies [2]. Its premise is that internal corporate users are trusted, but are restricted to use *safe* protocols when accessing `SCADA`. External users are allowed access only to content that is explicitly made public. The policy has the following goals:

- `Corp` hosts can access the `Oracle` servers and the HTTPS server in `SCADA`. They can also access all HTTP, HTTPS, DNS resources on the Internet.
- `SCADA` hosts can access Web, Email and DNS servers on `Corp`. Additionally, they can perform file transfer using FTP and HTTP with respective `Corp` servers.
- External hosts can access the HTTP, FTP and Email servers in `Corp`.
- R1, R2 can be managed from `Corp` using HTTPS and SSH. R2 can be managed from `SCADA` using SSH. R1 can also be managed from R2 using SSH.
- Firewall log messages are stored in a Syslog server located in `Corp`.
- OSPF is enabled across the entire site.

8.2 Implementation

A partial snippet of the *ForestFirewalls* high-level descrip-

```
 1  // library files
 2  import system.services.iana_services;
 3  import system.services.iana_icmp;
 4
 5  // zone−conduit security topology
 6  load_zone_conduit_model ‘‘zone_conduit.graphml’’
 7
 8  // define zone groups
 9  zone_group all_zones { z1,z2,z3,az1,fwz1,fwz2,fwz3
         }
10  zone_group scada_zone { z3 }
11  zone_group corp_zone { z1 }
12  zone_group internet_zone { z2 }
13  zone_group all_firewall_zones { fwz1, fwz2, fwz3 }
14  zone_group all_internal_zones { all_zones
15                                  \ internet_zone }
16
17  // FTP passive mode
18  port_group ftp_data_ports { 24500−24600 }
19  service ftp_data { protocol=tcp;
20                     tcp.dest_port=ftp_data_ports; }
21
22  // service groups
23  service_group ftp { iana_services.ftp_control,
24                      ftp_data }
25  service_group web { iana_services.http,
26                      iana_services.https }
27  service_group ping { iana_icmp.icmp_echo,
28                       iana_icmp.icmp_echo_reply }
29  service_group dns { iana_services.dns_tcp,
30                      iana_services.dns_udp }
31  service_group file_transfer { iana_services.http,
32                                ftp }
33
34  // policy rules
35  policy_rule file_transfer_rule {
36         scada_zone −> corp_zone : file_transfer }
37
38  policy_rule ping_rule {
39        corp_zone <−> scada_zone : ping }
40
41  policy_rule dns_rule {
42       scada_zone −> corp_zone : dns }
43
44  policy_rule web_rule {
45       scada_zone −> corp_zone : web }
46
47  // define policy
48  policy company_policy { file_transfer_rule,
49                  ping_rule, dns_rule, web_rule }
```

Listing 3: *ForestFirewalls policy description (partially shown).*

tion implementing the above policy goals is depicted in Listing 3. At the start, the required library files containing the predefined lists of IANA well known services are imported. Next, the Zone-Conduit security model is supplied as a GraphML file. The zones within this model can be grouped as necessary (line 9-15), to simplify the specification process and increase readability. Additionally, we define custom port groups, services and service-groups as needed.

A passive mode FTP data service is declared (lines 18–20) as it's the best-practice [9, 34] approach to enable FTP through firewalls. Ping is also defined for connectivity tests. The high-level policy rules are defined to match policy goals listed earlier. Finally a policy object container is used to hold all the policy rules (line 48).

8.3 Procedure and Results

Post policy parsing, an Alloy export is generated containing 134 object by *ForestFirewalls* for verification.

Running the 'no_rule_overlaps' assertion (not shown) returns a counter-example, indicating potential inconsistencies in the high-level policy. Upon inspection of the counter-example details in Alloy (Figure 7), we see that rules enabling HTTP services (`ip_protocol=6`, `dest_port=80`) between zones `z3` and `z1`, initiate the overlap. The root cause is the `file_transfer` and `web` service-groups in the high-level policy (Listing 3, lines 31 and 25), both containing HTTP. Once this is rectified (remove HTTP from `file_transfer`), no further counter-examples are found by Alloy.

Listing 4 shows the ACL-allocation map for firewall `R1`, indicating how ACLs are assigned to each of `R1`'s firewall interfaces. Also partially shown are the generated vendor neutral ACL rules. Note the explicit *deny all* rule supplementing the explicit permit rules at the end. The step also outputs the Zone-Firewall and Zone-Conduit models of the input network as graphical output (Figure 8).

The Alloy exports for network-level ACL verification have 828 (`Service` and `ACLRule`) objects. Assertion checks here yield no counter-examples.

The device-specific configurations are rendered from the network-level policy, using vendor and device specific *Mako* templates. Mako is a template library written in Python [28], enabling fast and easy integration into *ForestFirewalls*.

The device-level configurations generated for the specified platforms (*i.e.,* Netkit and Cisco), were auto-deployed to a Netkit-based emulated network. Once the Netkit VMs booted up, the automated tests were executed. The emulation results confirmed services explicitly allowed by high-level policy was admitted by the firewalls correctly. Moreover, exhaustive port-scans via *nmap* and *tshark* confirmed that no additional services were allowed through.

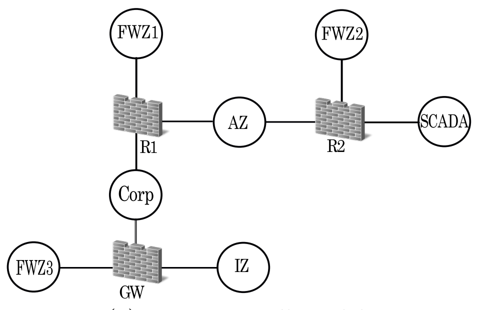
Post emulation testing, the device configurations were deployed to the real-network. Although we aim to automate this deployment, it is currently done manually as its not seen as an error-prone step in modern configuration tools [13, 14].

Once deployed, we re-executed the emulation test scripts on hosts in the various zones of the network. The tests confirmed that the services enabled by the input policy were passing across firewalls as expected. Supplementary port scans confirmed no additional services were allowed through.

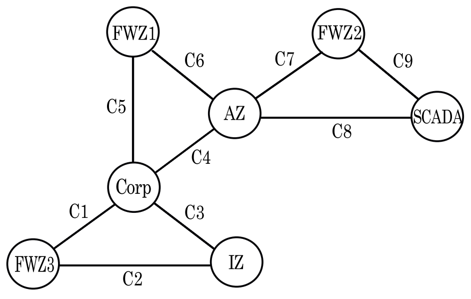
8.4 Performance analysis

A comparison of the firewall configurations observed in a previous original case study [2] and those generated by *ForestFirewalls* is shown in Table 1. We have distilled the core policy from the original case study by discussing with security consultants. Our target firewalls were Cisco ASA5505 and Linux IPtables devices instead of the Cisco IOS routers used originally. However, we are demonstrating and there are significant improvements achieved by our system.

As shown in Table 1, there are *no redundant ACLs* generated by *ForestFirewalls*. Each ACL serves a purpose and is assigned to an active firewall interface. Additionally, we observe that there are *no generic permit rules* generated by our system (*i.e.,* all-TCP, all-UDP or all-IP based rules). These rules produce security holes that can be exploited in cyber attacks. We also observe that there are *no intra-ACL rule interactions* in the ACLs generated by *ForestFirewalls*, making these configurations comparatively more efficient. These



(a) Zone-Firewall model.



(b) Zone-Conduit model of (a).

Figure 8: *System generated security models of the network.*

```
INFO Firewall—ACL map for firewall: R1
  Interface: 3(R1 to R2) direction: in ACL: acl_4
  Interface: 3(R1 to R2) direction: out ACL: acl_3
  Interface: 4(R1 to t1) direction: in ACL: acl_1
  Interface: 4(R1 to t1) direction: out ACL: acl_2

INFO Vendor neutral network—level ruleset for ACL: acl_2
  remark~enable corp_zone to scada_zone HTTPS traffic (return path)
  permit~tcp~from~10.0.0.16/29~to~10.0.0.0/29~sport~[443]~dport~['0—65535']~state~ESTABLISHED
  permit~tcp~from~10.0.0.16/29~to~10.0.128.4/30~sport~[443]~dport~['0—65535']~state~ESTABLISHED
  remark~enable scada_zone to corp_zone WEB traffic (forward path)
  permit~tcp~from~10.0.0.16/29~to~10.0.0.0/29~sport~['0—65535']~dport~[443]~state~NEW,ESTABLISHED
  permit~tcp~from~10.0.0.16/29~to~10.0.128.4/30~sport~['0—65535']~dport~[80]~state~NEW,ESTABLISHED
  deny~ip~from~any~to~any~sport~~dport~~state~
```

Listing 4: *System generated ACL-allocation map and network-level policy (partially shown, t1 is a router in Corp, comments are denoted by* remark*).*

Table 1: *High-level comparison of Original vs Generated configurations (LoC - Lines of Code).*

Type	Device-level LoC	Obsolete-ACL count	Generic permit-rule count	Intra-ACL interaction count
Original case study	2720	2	324	167
ForestFirewalls generated	714	**0**	**0**	**0**

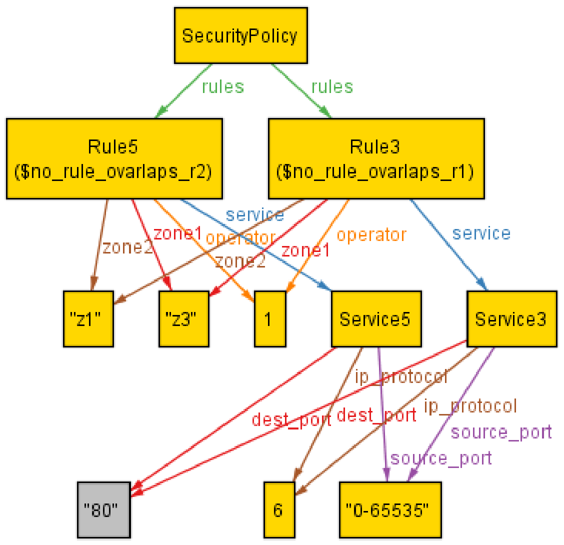


Figure 7: *Counter-example thrown by Alloy, indicating a high-level policy error.*

are almost obvious consequences of our design approach but note that the real firewalls [2] had all of these defects!

Our system only requires 68 high-level LoC (only 35 LoC are policy specific) to generate 714 device-level LoC to configure all 3 firewalls. A high-level policy with only 68 LoC has replaced 2720 error-prone, inefficient, device-level LoC of the original case study!

9. CONCLUSIONS AND FUTURE WORK

The current manual approach to firewall configuration is complex and error prone. Various firewall vendor tools attempt to facilitate high-level configuration, but these lack flexibility in specifying detailed traffic restrictions and do not reduce the configuration burden.

ForestFirewalls greatly reduces the configuration burden, and by use of high-level abstraction, templates and graphs, offers a simple and manageable approach to SCADA firewall configuration. Our system guarantees configuration accuracy through stage-wise validations employing SCADA best-practices, a formal verification tool (Alloy), and emulation based pre-deployment tests. The system gives users assurance of the generated device-level configurations delivering the expected firewall behaviour prior to deployment.

10. REFERENCES

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126, 2014.

[2] Anonymous. Identifying the missing aspects of the ANSI/ISA best practices for security policy, http://tinyurl.com/q4hjoxs.

[3] ANSI/ISA-62443-1-1. Security for industrial automation and control systems part 1-1: Terminology, concepts, and models, 2007.

[4] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *IEEE SSP '99*, pages 17–31. IEEE, 1999.

[5] BBC. Hack attack causes 'massive damage' at steel works, http://www.bbc.com/news/technology-30575104.

[6] S. Bellovin and R. Bush. Configuration management and security. *IEEE J. Sel. Areas Commun.*, 27(3):268–274, 2009.

[7] E. Byres. Using ANSI/ISA-99 standards to improve control system security. White paper, Tofino Security, May 2012.

[8] E. Byres. The air gap: SCADA's enduring security myth. *Communications of the ACM*, 56(8):29–31, 2013. http://cacm.acm.org/magazines/2013/8/166309-the-air-gap/fulltext.

[9] E. Byres, J. Karsch, and J. Carter. NISCC good practice guide on firewall deployment for SCADA and process control networks. *NISCC*, 2005.

[10] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Usenix Security*, 2006.

[11] Check Point. *NGX R65 CC Evaluated Configuration User Guide*. Check Point, software technologies Ltd., USA, 2008.

[12] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet security: Repelling the wily hacker*. Addison-Wesley, 2003.

[13] Cisco Systems. *Cisco ASA 5500 Series Configuration Guide using the CLI*. Cisco Systems, Inc., 170 West Tasman Drive, San Jose, CA 95134-1706, USA, 2010.

[14] Cisco Systems. Cisco ASA 5585-X adaptive security appliance architecture. White paper, Cisco Systems, May 2014.

[15] Cisco Systems. *Cisco Virtual Security Gateway for Nexus 1000V Series Switch Configuration Guide*. Cisco Systems, Inc., 170 West Tasman Drive, San Jose, CA 95134-1706, USA, 2014.

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. 46(9):279–291, 2011.

[17] K. D. Gourley and D. M. Green. Polygon-to-rectangle conversion algorithm. *IEEE COMP. GRAPHICS & APPLIC.*, 3(1):31–32, 1983.

[18] Graph Steering Committee. GraphML, 2003.

[19] S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: A slice abstraction for software-defined networks. In *ACM HotSDN '12*, pages 79–84. ACM, 2012.

[20] C. D. Howe. *What's Beyond Firewalls?* Forrester Research, Incorporated, 1996.

[21] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2011.

[22] R. Jamieson, L. Land, S. Smith, G. Stephens, and D. Winchester. Critical infrastructure information security: Impacts of identity and related crimes. In *PACIS*, page 78, 2009.

[23] Juniper Networks. *Firewall Filter and Policer Configuration Guide*. Juniper Networks, Inc., 1194 North Mathilda Avenue, Sunnyvale, California 94089, USA, 2011.

[24] S. Knight, H. Nguyen, O. Maennel, I. Phillips, N. Falkner, R. Bush, and M. Roughan. An automated system for emulated network experimentation. In *ACM CoNEXT '13*, pages 235–246. ACM, 2013.

[25] D. Levin, M. Canini, S. Schmid, and A. Feldmann. Panopticon: Reaping the benefits of partial SDN deployment in enterprise networks. *TU Berlin/T-Labs, Tech. Rep*, 2013.

[26] D. Libes. *Exploring Expect: A Tcl-based toolkit for automating interactive programs*. O'Reilly, 1995.

[27] A. X. Liu. Formal verification of firewall policies. In *IEEE ICC '08*, pages 1494–1498. IEEE, 2008.

[28] Mako. Mako templates for Python, http://www.makotemplates.org/.

[29] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 177–187. IEEE, 2000.

[30] J. Pearce. *Programming and Meta-Programming in Scheme*. Springer, 1998.

[31] M. Pizzonia and M. Rimondini. Netkit: Easy emulation of complex networks on inexpensive hardware. In *TRIDENTCOM '08*, page 7. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[32] A. D. Rubin and D. E. Geer. A survey of Web security. *Computer*, 31(9):34–41, 1998.

[33] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *ACM CoNEXT '14*, pages 213–226. ACM, 2014.

[34] K. Stouffer, J. Falco, and K. Scarfone. Guide to Industrial Control Systems (ICS) security. *NIST Special Publication*, 800(82):16–16, 2008.

[35] A. Wool. Architecting the Lumeta firewall analyzer. In *USENIX Security Symposium*, pages 85–97, 2001.

[36] A. Wool. A quantitative study of firewall configuration errors. *Computer, IEEE*, 37(6):62–67, 2004.

[37] A. Wool. Trends in firewall configuration errors: Measuring the holes in Swiss cheese. *Internet Computing, IEEE*, 14(4):58–65, 2010.

[38] H. Yan, D. A. Maltz, T. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4D network control plane. In *USENIX NSDI '07*, pages 369–382, 2007.

[39] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE SSP '06*, pages 15–213. IEEE, 2006.