Cross-Platform Firewall Orchestration for IoT Networks via OpenC2

Cross-Platform Firewall Orchestration for IoT Networks via OpenC2

Stefano Catenaro*, Daniele Canavese†, Daniele Bringhenti*, Gianmarco Bachiorrini*, Matteo Repetto†
*Politecnico di Torino, Torino, Italy
†CNR-IMATI, Genova, Italy
stefano.catenaro@studenti.polito.it, daniele.canavese@cnr.it, daniele.bringhenti@polito.it
gianmarco.bachiorrini@polito.it, matteo.repetto@cnr.it

***Abstract*—The rapid expansion of the Internet of Things (IoT) ecosystems introduces significant security challenges due to the device heterogeneity, resource constraints, and dynamic network topologies. Traditional host-based protections are often insufficient, necessitating network-level enforcement mechanisms such as firewalls. However, managing multiple, heterogeneous firewalls across distributed environments can be complex and fragmented. This paper presents an approach based on the Open Command and Control (OpenC2) standard for the unified orchestration of StateLess Packet Filtering (SLPF) systems in heterogeneous IoT infrastructures. The proposed solution centers on an OpenC2-compliant Actuator Manager that coordinates specialized components able to translate platform-independent OpenC2 commands into native firewall configurations. Our implementation supports Linux iptables, OpenStack Security Groups, Kubernetes Network Policies, and Microsoft Azure Network Security Groups, and provides rule persistence, scheduled execution, and centralized command management through a single unified control interface. We validated our solution through syntax and semantic checks, functional tests, and performance evaluations across diverse networks, demonstrating its effectiveness and efficiency in cross-platform enforcement.**

***Index Terms*—OpenC2, firewall, packet filtering, firewall orchestration, policy enforcement**

I. Introduction

The rapid proliferation of Internet of Things (IoT) devices is fundamentally transforming modern digital ecosystems, leading to highly distributed systems characterized by large scale, technological heterogeneity, and dynamic network topologies. At the same time, this evolution introduces significant security challenges. IoT environments are typically characterized by a large number of heterogeneous, resource-constrained devices, often deployed in untrusted or exposed networks. These characteristics limit the applicability of traditional host-based security mechanisms, leading to a shift in security enforcement toward network-level mechanisms, such as traffic filtering, applied outside the devices themselves. Such network-level approaches, however, are often specific to particular platforms or deployment environments. As a result, security management can become fragmented, inconsistent, and difficult to coordinate across heterogeneous devices and networks. This fragmentation limits the ability to mount coordinated, timely responses to security incidents, such as isolating compromised devices or preventing the spread of attacks.

To address these limitations, it is beneficial to adopt a higher-level firewall management system that can coordinate a heterogeneous fleet of filtering devices in a uniform and efficient manner, independent of their specific implementations. By abstracting platform-specific enforcement mechanisms behind a common control layer, such a system enables consistent policy specification, centralized orchestration, and coherent security responses across heterogeneous environments. Moreover, expressing firewall operations in a machine-readable format makes the management process amenable to automation, thereby reducing manual intervention, improving scalability, and facilitating timely reconfiguration in dynamic, distributed scenarios.

In this paper, we propose an approach that leverages the Open Command and Control (OpenC2) standard to provide a unified, automated interface to multiple Stateless Packet Filtering (SLPF) firewalls deployed in a network. Our implementation uses a central component, the Actuator Manager, which coordinates multiple distributed Actuators, each configured to monitor a specific SLPF firewall. Each supported actuator allows the system to encapsulate technology-specific enforcement logic while preserving a uniform control interface. Under the hood, our system translates implementation-agnostic OpenC2 commands into concrete firewall configurations across multiple technologies, including Linux iptables, OpenStack Security Groups, Kubernetes Network Policies, and Microsoft Azure Network Security Groups (NSGs).

The contributions of our paper are the following:

- an OpenC2/SLPF-compliant actuator-manager architecture for coordinating heterogeneous stateless packet-filtering backends through a common command-and-response model;
- a set of backend-specific translators that map OpenC2/SLPF commands to native abstractions in Linux iptables, OpenStack Security Groups, Kubernetes Network Policies, and Azure Network Security Groups;
- a command-lifecycle management mechanism supporting validation, dispatching, scheduling, persistence, response generation, and explicit reporting of unsupported backend capabilities;
- an experimental validation of the implemented backends, covering syntactic correctness, semantic enforcement, functional behavior, and execution overhead.

The full source code is freely available online[1].

The remainder of this paper is structured as follows. Section II presents a selection of related works, while Section III introduces the basic concepts behind OpenC2. Section IV is the core of this paper and contains the description of our approach. Finally, Section V presents the validation of our methodology, and Section VI concludes this article with some final remarks and directions for future avenues.

II. Related works

Security in IoT environments has been extensively investigated, highlighting the challenges of deploying traditional host-based protection mechanisms on resource-constrained, heterogeneous devices [1]. As a result, network-level defenses, such as traffic filtering, segmentation, and device isolation, are widely recognized as effective tools to mitigate attack propagation and enhance the resilience of large-scale IoT deployments. However, the scientific literature is scarce on this topic, and existing approaches are frequently evaluated within specific infrastructures or single administrative domains, leaving opportunities for more coordinated enforcement across heterogeneous environments.

Harish et al. presented an enforcement architecture for edge-cloud IoT environments based on MUD (Manufacturer Usage Description) that moves policy enforcement away from individual local networks and into programmable edge switches [2]. The paper proposes first identifying IoT devices and their types, then applying the corresponding MUD rules via a P4-based data plane, with packet marking used to distinguish device traffic, a Bloom-filter-based mechanism to handle reverse flows, and a decision tree classification strategy to improve scalability on programmable switches. Although conceptually similar to our approach, this paper still relies on a technology-independent language (MUD) to enforce traffic in an IoT network.

Hoang et al. consider the management of multiple firewalls in virtualized environments, where a tenant may operate several virtual networks, each containing multiple enforcement points [3]. To address the resulting policy-management complexity, they propose treating all firewalls for a tenant as a single meta-firewall, enabling administration through a unified abstraction rather than separate device-level configurations.

Recent works in automated network security configuration further explore orchestration mechanisms for firewall and security policy management in virtualized and software-defined environments. For instance, Bringhenti et al. provide a comprehensive survey of automation techniques for network security configuration, highlighting how existing approaches often rely on environment-specific abstractions and lack a unified control model across heterogeneous enforcement technologies [4].

Kovačević et al. provide a systematic review of the research on automatic translation from high-level security policies to low-level firewall rules, offering a useful conceptual map of the field rather than a single implementation [5]. Their survey covers more than 20 approaches and shows that most existing solutions rely on specialized policy languages that are subsequently compiled into firewall rule sets, sometimes with support from formal methods, ontologies, or graphical models. At the same time, the authors observe that, despite notable progress in abstraction and readability, many limitations still hinder broader practical adoption.

Despite significant progress in firewall orchestration, high-level policy translation, and IoT security enforcement, most existing solutions remain constrained to specific domains or enforcement technologies. SDN-centric approaches focus on flow-level programmability, survey-based frameworks emphasize conceptual classification rather than implementation interoperability, and IoT-oriented solutions are typically tailored to specific device types or network paradigms. As a result, there is still a lack of unified approaches capable of coordinating heterogeneous packet-filtering mechanisms across diverse environments. Our work builds on these efforts by providing a standardized, technology-agnostic control plane for multiple filtering backends in IoT environments.

III. OpenC2

Open Command and Control (OpenC2) is an open, vendor-neutral standard developed by OASIS to enable interoperable and automated command and control of cyber defense components across heterogeneous environments [6].

The OpenC2 architecture follows a producer–consumer model, where a producer generates commands, and one or more consumers execute them on the underlying security technology, returning a response containing the command status and any associated results. A consumer can be an *actuator*, a component usually responsible for managing a single device (e.g., a firewall). An actuator receives OpenC2 commands and translates them into device-specific abstractions of the managed systems, enabling uniform control across heterogeneous technologies. Alternatively, a consumer can also be an *actuator manager*, an intermediary component that coordinates one or more actuators.

An OpenC2 command is expressed in a structured and machine-readable format, typically serialized in JSON. It includes an action, which specifies the operation to be performed (e.g., `allow` or `deny`), a target, identifying the object of such operation, an optional actuator, specifying the system intended to process the command, and multiple optional arguments, which refine the execution context, such as timing or response requirements. OpenC2 does not mandate a specific transport protocol for command and reply traffic. Instead, transfer specifications define how commands are conveyed over standard mechanisms such as HTTPS [7] and MQTT [8].

The allowed actions, targets, parameters, and results for a producer-consumer interaction are defined by a *profile*, which, in a nutshell, defines the communication interface with one or more actuators. OASIS has already standardized a series of profiles, such as the SLPF one that specializes the OpenC2 language for the management of stateless packet filtering functions [9].

[1]https://github.com/mattereppe/otupy (accessed: 27/05/2026).

TABLE I
SUPPORTED ACTIONS OF THE SLPF PROFILE.

ACTION	TARGET	DESCRIPTION
`allow`	connection/network	allow some traffic
`delete`	rule number	delete a rule
`deny`	connection/network	deny some traffic
`query`	features	return the valid action–target pairs
`update`	file	update the rules with the file's content

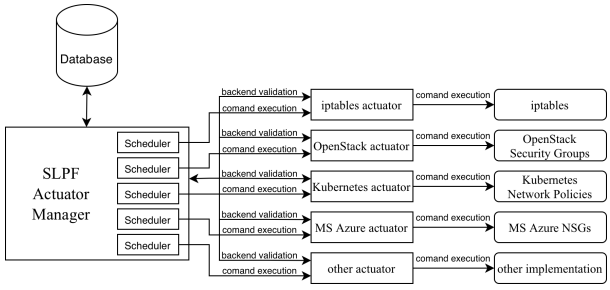

Fig. 1. Architecture of our approach.

The SLPF profile defines the actions shown in Table I.

The profile supports creating allow and deny rules based on the usual source/destination IP addresses and port numbers. In addition, it supports deleting existing rules and updating an entire rule set using a configuration file. Additionally, it provides several additional parameters that enable temporal execution of each rule (e.g., when a rule should begin or stop being enforced), their persistency (to keep them across device reboots), insert_rule (to control rule placement within a rule set), and how blocked traffic should be handled (i.e., send an ICMP host unreachable to the source).

IV. OUR APPROACH

Our approach centers on an actuator manager that serves as a dispatcher for a set of SLPF actuators. Within this architecture, communication is expressed through OpenC2 commands and complies with the OASIS SLPF profile, thereby ensuring a consistent, interoperable control model across heterogeneous enforcement environments. Fig. 1 reports the general architecture of our solution.

The actuator manager and its associated actuators were developed as an extension of Otupy [10], a flexible Python framework for implementing OpenC2-based control of security functions, SQLite3[2] for managing database connections, and APScheduler[3] for scheduling the OpenC2 commands.

Implementing OpenC2/SLPF orchestration over heterogeneous firewalls required addressing several semantic and operational mismatches. First, the SLPF profile defines abstract actions such as allow or delete, whereas each backend exposes different primitives, rule models, and default enforcement semantics. Second, the implementation must preserve OpenC2 command semantics while avoiding unsafe emulation of unsupported features. Third, command execution requires consistent handling of rule identifiers, rule ordering, persistence, delayed execution, and backend-specific error reporting. Our proposed actuator manager architecture addresses these challenges by separating common OpenC2 command processing from backend-specific translation logic.

The following paragraphs detail the inner workings of our approach and its components.

A. Actuator manager

The *actuator manager* operates as a super-class implementing all core functionalities required by an SLPF-compliant actuator, while back-end-specific actuators extend it to translate OpenC2 commands into the native syntax and semantics of the underlying firewall technology. The Actuator Manager implements the common OpenC2/SLPF command-processing interface, including command validation, scheduling, persistence management, response generation, and dispatch to backend-specific actuators. However, the semantic support of individual SLPF actions depends on the capabilities of each underlying packet-filtering technology. Unsupported actions or parameters due to backend-specific limitations are explicitly reported via not-supported responses rather than silently emulated.

This architectural separation allows common mechanisms, such as command validation, rule persistence, database interaction, command scheduling, and response generation, to be implemented once and reused across all back-end modules. Consequently, integrating additional filtering systems requires only implementing platform-specific logic, including mapping OpenC2 targets and action-specific arguments to native firewall entities, and overriding action-specific execution methods.

The actuator manager maintains an internal database that keeps a consistent rule state by storing active filtering rules, their timing and persistence parameters, and any back-end-specific execution data. This persistent storage enables accurate rule reconstruction and proper life-cycle management, even on firewall systems that do not support rule persistence after a reboot. The database layer is entirely managed by the actuator manager, allowing back-end implementations to rely on a unified infrastructure without re-implementing storage logic.

Within the actuator manager, internal schedulers manage the timed execution of OpenC2 commands. Since the SLPF profile supports timing arguments, commands may require delayed activation or automatic revocation. The actuator manager assigns a dedicated scheduler to each back-end actuator and oversees its operation, ensuring the correct execution life-cycle of commands within the corresponding environment. To prevent race conditions and inconsistent rule states, schedulers enforce serialized execution within the same back-end. Additionally, information about scheduled but not executed commands is stored in the database at system shutdown, allowing pending operations to be restored during actuator initialization.

[2]https://sqlite.org/ (accessed: 27/05/2026).

[3]https://apscheduler.readthedocs.io/ (accessed: 27/05/2026).

The actuator manager operates in two distinct modes: a DB (DataBase) mode and a file mode. In DB mode, rules generated by allow and deny commands are stored in the internal database and synchronized with the back-end implementation. When an update command is received, all active rules are removed, then the manager switches to file mode, and the rules defined in the specified file are applied. Conversely, if an allow or deny command is received while operating in file mode, the actuator manager clears the file-based configuration, switches to DB mode, and resumes rule management through the database. This dual-mode mechanism guarantees coherent state transitions and prevents inconsistencies between DB-based and file-based configurations.

For each received OpenC2 command, the Actuator Manager follows a structured processing workflow. First, the command is validated against the OpenC2 Language Specification [6], and then against its specific back-end, allowing an early failure if it does not meet the specifications. Then, depending on the action, the filtering rules are recorded in the internal database together with any custom back-end data.

During startup, the actuator manager initializes the database, restores persistent rules when necessary, and instantiates the schedulers. Any previously scheduled, not executed commands are reloaded at startup, ensuring operational continuity. Conversely, during shutdown, non-persistent rules are removed, persistent rules are saved when required, and scheduled commands are stored for restoration at the next startup.

B. iptables actuator

The iptables actuator for Linux interfaces with the native `iptables` and `ip6tables` frameworks for IPv4 and IPv6 packet filtering. The actuator manages packet filtering by creating custom chains for both IPv4 and IPv6 traffic, structured to handle input, output, and forward flows. These chains are linked to the main system chains, enabling controlled and isolated rule management while preserving compatibility with existing configurations.

The allow and deny actions are translated into iptables rules with `ACCEPT`, `DROP`, or `REJECT` iptables targets depending on the received command, while delete actions remove the corresponding rules from the appropriate chains. Rules are applied to the `INPUT` chain for ingress, the `OUTPUT` chain for egress, or both for bidirectional traffic, and are always added to the `FORWARD` chain to handle forwarded packets. Rule insertion respects the first-match processing model, ensuring correct ordering and predictable behavior.

The update action validates and loads rule sets from configuration files, replacing the current firewall configuration with the specified rule set for both IPv4 and IPv6.

Active rules are saved and restored at shutdown and startup using native save and restore utilities (i.e., `iptables-save`, `iptables-restore`, `ip6tables-save`, and `ip6tables-restore`).

The Linux back-end fully supports the SLPF profile semantics, except for the option to drop the traffic and send a false acknowledgment.

C. OpenStack actuator

The OpenStack back-end interfaces with the OpenStack networking service to manage Security Groups and their security rules. Each OpenC2 command targeting IPv4 or IPv6 traffic is enforced by dynamically creating or updating Security Groups associated with the relevant project ports and inserting the corresponding security rules within those groups.

For `allow` actions, the actuator translates the OpenC2 target parameters (source and destination addresses, protocol, and port numbers) into the corresponding OpenStack security rule fields. The actuator assigns the security group ID based on the ports affected by the command, and attaches those ports to the corresponding security groups. Conversely, deletion commands remove the specified security rules from the associated Security Groups, disconnecting ports and removing the group entirely if it becomes empty.

Due to OpenStack's default deny policy, the `deny` action is not supported. In addition, the `update` action cannot be implemented as OpenStack does not provide a mechanism to update firewall configurations through rule sets. As a result, both commands return a not-implemented response.

Despite these limitations, OpenStack security rules are natively persistent, so there is no need to manage external rule files to preserve configurations across system restarts.

D. Kubernetes actuator

The Kubernetes back-end leverages the Kubernetes networking framework to manage network policies within a cluster. OpenC2 commands are enforced by dynamically generating labels for the subnet or connection specified in the command and applying them to the affected pods. These labels are then referenced by the Network Policies created to implement the required filtering behavior.

For `allow` actions, the actuator translates the OpenC2 command into one or more network policy objects, while deletion commands remove the corresponding network policies from the namespace. If a removed policy was the last one associated with a label, that label is also removed from the affected pods. Analogously, the `update` action replaces the active policies within the namespace using a YAML file provided as the command target.

Because Kubernetes enforces a default deny behavior, the `deny` action is not supported, and the actuator returns a not-implemented response when such commands are received.

Network Policies in Kubernetes are natively persistent and remain active across system restarts without requiring external rule files.

E. Azure actuator

The Microsoft Azure back-end uses the Azure Network Security Groups (NSGs) to manage security rules.

For `allow` and `deny` actions, OpenC2 commands are translated into security rules by mapping each parameter to its corresponding Azure field. Deletion commands remove the corresponding security rules, ensuring that both inbound and

outbound rules are deleted when the OpenC2 direction is set to both.

The `update` action is not supported because Azure does not provide a mechanism to load firewall configurations from external files. However, security rules are natively persistent, eliminating the need for additional file-based storage or restoration mechanisms.

V. Validation

This section presents the validation of our approach, aimed at verifying the correctness, functionality, and performance of its implementation across the supported firewall technologies. The Azure back-end was not tested, as no suitable environment was available to support its evaluation. Our validation process follows a three-phase approach: syntactic and semantic validation, functional testing, and performance evaluation. All tests were implemented using the `pytest` framework[4], enabling automated execution and complete reproducibility of the results.

A. Syntactic and semantic validation

The syntactic and semantic validation phase verifies that OpenC2 commands are correctly formatted, interpreted, and processed in accordance with the OpenC2 Language Specification and the standard SLPF profile. We generated a test input set by constructing all possible commands that could be issued to the actuator manager, accounting for every supported action and all valid combinations of targets and arguments. Commands were classified as valid or invalid based on the constraints defined by the SLPF specification. Validation was performed by serializing and deserializing commands and responses, comparing the resulting representations with the original JSON messages, and verifying message exchanges between producer and consumer over HTTP. At the end, we obtained a 100% success rate.

B. Functional testing

Functional testing evaluates each actuator's ability to correctly translate SLPF commands into concrete enforcement actions within its respective environment. Dedicated IoT test environments were deployed for each supported back-end, including virtual machines for the iptables implementation and cloud-based environments for the remaining actuators. Network traffic was generated using `hping3` and captured with `tcpdump`, while Wireshark[5] was used to analyze packet traces and verify whether traffic was correctly allowed or blocked according to the applied filtering rules.

C. Performance evaluation

Performance evaluation measures the responsiveness and efficiency of the system by analyzing both producer-side latency and consumer-side execution times. On the producer side, latency is defined as the time elapsed between the dispatch of an OpenC2 command and the reception of the corresponding response. This metric captures the overall communication overhead introduced by the system, including serialization, network transmission, and message handling. On the consumer side, we measure the execution time within the actuator infrastructure. This includes the total time required to fully apply a command, from its reception by the Actuator Manager to the completion of the corresponding enforcement action on the target firewall. Additionally, we isolate the actual backend execution time, such as rule insertion or deletion, to better assess the efficiency of each underlying filtering technology.

We used the following virtualized test beds:

- the iptables back-end was evaluated in a lightweight virtualized environment composed of two VirtualBox[6] virtual machines connected to the same virtual network, thereby allowing direct end-to-end communication between them;
- the OpenStack back-end was tested in a cloud-based environment consisting of three virtual machines, where one node acted as the external access point and two internal instances served as the traffic endpoints on which security-group operations were applied;
- the Kubernetes back-end was deployed on the same OpenStack-based cluster architecture, but the measurements focused on containerized workloads running inside pods distributed across the two internal nodes.

TABLE II
Producer-side results of our performance evaluation.

Metric	Time $[s]$		
	iptables	Kubernetes	OpenStack
number of runs	142	113	113
`allow`: total time	0.005 ± 0.001	0.601 ± 0.365	1.662 ± 0.119
`delete`: total time	0.004 ± 0.001	0.004 ± 0.001	0.004 ± 0.001

TABLE III
Consumer-side results of our performance evaluation.

Metric	Time $[s]$		
	iptables	Kubernetes	OpenStack
number of runs	142	113	113
`allow`: total time	0.035 ± 0.003	0.682 ± 0.367	2.718 ± 0.200
`allow`: exec. time	0.033 ± 0.003	0.084 ± 0.010	1.056 ± 0.085
`delete`: total time	0.083 ± 0.011	0.576 ± 0.428	1.172 ± 0.406
`delete`: exec. time	0.082 ± 0.010	0.574 ± 0.428	1.171 ± 0.406

Table II shows the results of our tests on the producer side, where the values after $\pm$ are the standard deviations.

The *producer-side total time* is the interval between sending an OpenC2 command and receiving its response.

Table III reports instead the results of our tests on the consumer side, where the values after $\pm$ are the standard deviations.

We observe that producer-side latencies are significantly shorter than consumer-side execution times. This behaviour, however, is expected, as the actuator executes the command

[4]https://docs.pytest.org/ (accessed: 27/05/2026).

[5]https://www.wireshark.org/ (accessed: 27/05/2026).

[6]https://www.virtualbox.org/ (accessed: 27/05/2026).

asynchronously in the background and returns a response to the producer before the underlying operation has actually completed.

Our experimental results highlight a clear performance hierarchy among the three back-ends we tested. iptables emerges as the most efficient and, more importantly, the total execution time on the consumer side almost coincides with the actual rule enforcement time, indicating that the OpenC2 abstraction introduces negligible overhead. OpenStack, by contrast, exhibits the highest latency, with average values generally exceeding one second for most metrics. This result suggests that the main performance bottleneck lies not only in rule deployment itself, but also in the translation of the OpenC2 command into the OpenStack model, including the creation of Security Groups. Kubernetes represents an intermediate case, yet with substantially better performance than OpenStack, since average times remain within approximately half a second. This indicates good responsiveness in the management of Network Policies, although a non-negligible translation cost remains due to the need to identify pods and generate the corresponding labels and policies.

VI. Conclusions

This paper presented an OpenC2-based approach for cross-platform firewall orchestration in heterogeneous IoT environments. By extending the Otupy framework with an SLPF Actuator Manager and a pool of technology-specific actuators, our proposed solution enables unified and automated enforcement across multiple packet-filtering back-ends. The experimental results confirmed both the correctness and the feasibility of the approach, while also showing that performance strongly depends on the underlying platform: iptables achieved the lowest latency, Kubernetes provided a good balance between flexibility and responsiveness, and OpenStack exhibited the highest overhead due to its more complex translation and deployment process. Overall, our study demonstrates that OpenC2 is a viable abstraction layer for interoperable firewall management in heterogeneous IoT infrastructures.

Future work will focus on extending support to additional enforcement technologies, improving translation efficiency on higher-latency platforms, and evaluating the framework at a larger scale.

Acknowledgment

This project has received funding from the European Union's Horizon Europe Research and Innovation Programme under grant agreement No. 101168144. The views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or the European Cybersecurity Competence Centre. Neither the European Union nor the granting authority can be held responsible.

References

[1] A. Sharma and K. Bhushan, "A comprehensive survey on IoT security: Challenges, security issues, and countermeasures," *Computer Science Review*, vol. 59, p. 100839, 2026.

[2] S. A. Harish, H. Kothapalli, S. Lahoti, K. Kataoka, and P. Tammana, "IoT MUD enforcement in the edge cloud using programmable switch," in *Proceedings of the FFSPIN 2022: Workshop on Formal Foundations and Security of Programmable Network Infrastructures*, ser. FFSPIN '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–7.

[3] X. T. Hoang, V. S. Pham, and N. D. Bui, "An Efficient Administration for Multiple Firewalls in Cloud Environments," in *Proceedings of ICISN 2024: International Conference on Intelligent Systems and Networks*, T. D. L. Nguyen, M. Dawson, L. A. Ngoc, and K. Y. Lam, Eds. Singapore: Springer Nature, 2024, pp. 1–10.

[4] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Computing Surveys*, vol. 56, no. 3, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3616401

[5] I. Kovačević, B. Štengl, and S. Groš, "Systematic review of automatic translation of high-level security policy into firewall rules," in *Proceedings of MIPRO 2022: International Convention on Information, Communication and Electronic Technology*, 2022, pp. 1063–1068.

[6] "Open Command and Control (OpenC2) Language Specification Version 1.0," OASIS Open, Committee Specification 02, 2019, 24 November 2019. [Online]. Available: https://docs.oasis-open.org/openc2/oc2ls/v1.0/cs02/oc2ls-v1.0-cs02.html

[7] "Specification for Transfer of OpenC2 Messages via HTTPS Version 1.1," OASIS Open, Committee Specification 01, 2021. [Online]. Available: https://docs.oasis-open.org/openc2/open-impl-https/v1.1/cs01/open-impl-https-v1.1-cs01.html

[8] "Specification for Transfer of OpenC2 Messages via MQTT Version 1.0," OASIS Open, Committee Specification 01, 2021. [Online]. Available: https://docs.oasis-open.org/openc2/transf-mqtt/v1.0/cs01/transf-mqtt-v1.0-cs01.html

[9] "Open Command and Control (OpenC2) Profile for Stateless Packet Filtering Version 1.0," OASIS Open, Committee Specification 01, 2019, 11 July 2019. [Online]. Available: https://docs.oasis-open.org/openc2/oc2slpf/v1.0/cs01/oc2slpf-v1.0-cs01.html

[10] M. Repetto, "Otupy: A flexible, portable, and extensible framework for remote control of security functions," *Computer & Security*, vol. 158, 2025.