IoTWall: An Efficient Host-Based Firewall for Resource-Constrained IoT Devices

Markus Schramm*, Lukas Bechtel*, Florian Hoss*, Michael Menth†, Tobias Heer*
*Esslingen University, Germany, {markus.schramm, lukas.bechtel, flhoit00, tobias.heer}@hs-esslingen.de
†University of Tübingen, Germany, menth@uni-tuebingen.de

***Abstract*—In areas such as the industrial sector, IoT devices have become an important component. For example, they serve as temperature sensors or switches. Often, IoT devices share the wireless network with other devices, increasing the potential for attacks through compromised hosts in the network. IoT devices typically lack a packet filtering mechanism that network administrators can configure to limit access to the IoT device by trusted hosts. To enhance IoT device security, we present *IoTWall*, a lightweight host firewall designed to run on resource-constrained IoT devices. IoTWall is easy to integrate into existing IoT software projects with only a few code changes to encourage developers to adopt stronger security measures. A REST API enables easy configuration by network administrators without requiring code changes. We also show that IoTWall operates efficiently within the constraints of resource-constrained devices with acceptable latency and energy consumption.**

***Index Terms*—IoT, Network Security, Microcontroller, ESP32**

I. Introduction

IoT devices have become an important component in many professional applications, such as the industrial sector and building automation. For example, in building automation, presence detectors offer novel interaction methods and improve safety. In the industry, smart screwdrivers report the torque for traceability of manufacturing quality.

These IoT devices are a popular target for attackers, e.g., due to delayed or missing updates. An example of common threats to IoT devices are botnets, such as Mirai. To protect devices from malicious hosts on the Internet, network administrators deploy network firewalls. Additionally, devices in the network, such as servers and workstations, have a host firewall to protect against potentially malicious hosts in the same network. IoT devices usually lack such a host firewall. Moreover, to efficiently configure a large number of IoT devices, IT departments need a standardized configuration interface. Therefore, in this paper, we present the design and implementation of a host firewall that runs on IoT devices, called *IoTWall*. The implementation supports the ESP32 microcontroller by Espressif and uses their ESP-IDF toolchain. As the ESP32 uses the lwIP library [1] as network stack, *IoTWall* uses interfaces that lwIP provides. We design *IoTWall* specifically for IoT devices integrated into large IT infrastructures. The use of *IoTWall* should not require additional hardware or large code changes. These requirements lead to the following design goals for *IoTWall*:

- **G1** Easy integration of the host firewall into source code of existing IoT applications with few code changes
- **G2** Ruleset configuration by network administrators via a standard interface and ruleset persistence between restarts
- **G3** Small size of the compiled application and the ruleset to keep the required flash memory size small
- **G4** Computational efficiency to have a low impact on the battery lifetime of IoT devices

We begin this paper by discussing related work. Next, we present the design and implementation of *IoTWall*. We publish the implementation on GitHub[1] and contribute missing functionality to the lwIP project. In the evaluation, we show that the implementation meets the design goals of small size and computational efficiency, making the design applicable to IoT devices. Finally, we conclude the paper.

II. Related Work

Existing host firewalls for IoT devices have disadvantages that we solve with the design goals defined in Section I: They cannot be easily integrated into existing IoT applications, do not provide a configuration interface for network administrators, or are not available for broad use.

Pahl et al. [2] developed a traffic monitor for IoT devices to learn communication relations and filter disallowed communication. While they run the traffic monitor on Raspberry Pis, we focused on efficient filtering on less powerful microcontrollers. Maheshwari et al. [3] present a firewall that is independent of the network stack. This has the advantage that the firewall is compatible with various microcontrollers. McManus [4] presents *ESP8266-Firewall*, a firewall for Arduino microcontrollers. Both firewalls do not have a configuration interface for network administrators, and developers cannot easily integrate them into existing applications with only few code changes.

Commercial security solutions for IoT devices are offered by wolfSSL [5], Check Point [6], and Sectigo [7]. *wolfSentry* by wolfSSL is open source and the most similar project to IoTWall, as it uses lwIP. Its features exceed the functionality of a firewall, making the configuration more complicated. Moreover, wolfSentry uses a different approach than IoTWall to intercept packets from lwIP, requiring many changes in the lwIP code and a self-compiled lwIP version. This complicates the integration into existing IoT applications. The firewall

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 528745080 - FIP 68 and by the project "FHP: Qualifizierung und Entwicklung des professoralen Personals der Hochschule Esslingen für zukunftsweisende Themen" (FKZ: 03FHP115), funded by the BMBF and MWK Baden-Württemberg. The authors alone are responsible for the content of the paper.

[1]https://github.com/hs-esslingen-it-security/IoTWall

Accepted for publication in Proceedings of *IEEE International Conference on Factory Communication Systems (WFCS)*, June 10-13, 2025

solutions from Check Point and Sectigo are only commercially available. In contrast, IoTWall is open source, offering an easy solution to enhance IoT device security.

III. Design and Implementation of IoTWall

In this section, we highlight the design of IoTWall. The entire design focuses on the design goals **G3** and **G4**, i.e., small application and ruleset size and computational efficiency. This keeps the memory and CPU requirements on the microcontroller small, such that IoTWall runs on a wide range of devices without additional cost due to hardware upgrades. First, we explain the packet filtering mechanism, focusing on design goals **G3** and **G4**. Second, we explain the interaction of IoTWall with lwIP, solving design goal **G1**. Lastly, we explain the configurability and persistence, solving the design goal **G2**.

A. Packet Filtering Mechanism

Typically, firewalls filter packets based on a list of rules. Each rule contains the parameters of the IP header five-tuple. A parameter can contain a specific value, a range, or a wildcard. For example, allowing an IoT device to communicate only with a single server results in a specific value for the IP address, whereas communication with a complete machine network results in a range of IP addresses. If all parameters of a rule match an incoming or outgoing packet, the action attached to this rule is executed, i.e., the packet is either accepted or dropped. If no rule matches a packet, the firewall executes the default rule. Table I shows an example ruleset of IoTWall with specific values, ranges, and wildcards.

To increase the filtering efficiency, rules of IoTWall have only one IP address and do not differentiate between source and destination. This differentiation is typically required for routers that forward packets. Since IoT devices usually do not forward packets, we save this unnecessary comparison. Similarly, we assume that resource-constrained IoT devices do not implement any network and transport layer protocol other than *TCP*, *UDP*, and *ICMP*. Hence, we limit the implementation to these three protocols to reduce application size. Finally, we only support the two most important actions, *accept* and *drop*.

B. Interaction with lwIP

To enable the integration of IoTWall into existing IoT applications with only few code changes, developers should not need to change code that is responsible for receiving and sending packets. For example, in existing host firewalls [3, 4], developers need to implement calls to the filtering function at each position where the application sends and receives packets. *lwIP* offers the mechanism of hooks to execute functions when receiving and sending packets. An

TABLE I
Example ruleset on IoTWall

IP Address	Protocol	Src Port	Dst Port	Action
1.1.1.0/24	TCP	*	443	Accept
2.1.1.20	UDP	49,152 - 65,535	1194	Accept

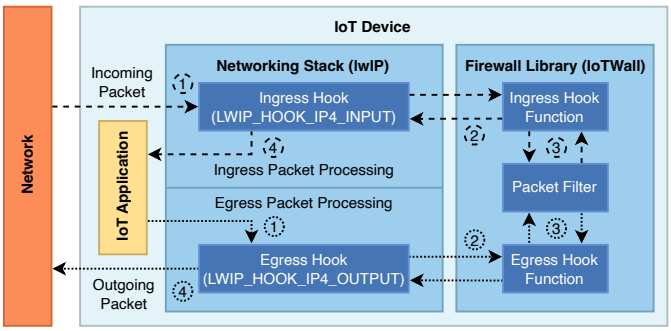


Fig. 1. Interaction of lwIP with IoTWall and the IoT application. The numbers show in which order a packet traverses the components.

IoT application can register a hook function to each hook that lwIP provides, respectively. Specifically, IoTWall uses the hooks `LWIP_HOOK_IP4_INPUT` to invoke packet filtering for incoming packets and `LWIP_HOOK_IP4_OUTPUT` for outgoing packets, respectively. For IPv6 packets, IoTWall uses the hooks `LWIP_HOOK_IP6_INPUT` and `LWIP_HOOK_IP6_OUTPUT`. This way, IoTWall can process the packets before they arrive at the IoT application and after the IoT application sends them without modifying existing code. In the current *lwIP* version, only the input hooks exist. We contribute the output hooks to the *lwIP* repository.

Figure 1 shows how lwIP interacts with IoTWall and the IoT application when the IoT application sends or receives IPv4 packets. When a packet arrives, lwIP processes it and calls the ingress hook function that IoTWall registered. IoTWall calls its packet filtering function that checks with the ruleset whether the packet is allowed. The packet filtering function returns to the hook function with a return value indicating whether the packet is allowed. Finally, the hook function returns the value to lwIP. If the packet is allowed, lwIP passes the packet to the application. Otherwise, lwIP drops the packet.

Developers that integrate IoTWall into their IoT application need to register the hook functions for the four mentioned hooks by adding a few lines to the IoT application code. They are not required to modify existing code. Listing 1 demonstrates how to register the function `lwip_hook_ip4_input` as a hook function for the `LWIP_HOOK_IP4_INPUT` hook. In the first line, the hook function is declared. The hook function is registered in the second line by referencing it in the `LWIP_HOOK_IP4_INPUT` macro. lwIP uses this macro to call the hook function when an IPv4 packet arrives. The hook function is registered at compile time. lwIP does not support changing the hook function while the IoT application runs, which is no problem for IoTWall.

```
int lwip_hook_ip4_input(struct pbuf *pbuf, struct netif *
    input_netif);
#define LWIP_HOOK_IP4_INPUT lwip_hook_ip4_input
```

Listing 1. IPv4 input hook function registration

C. Configurability and Persistence

Network administrators should be able to configure the firewall ruleset in the field without changing the source code.

Therefore, IoTWall integrates a REST API with four different endpoints to get, insert, update, and delete rules. We include the API schema in the GitHub repository.

The access to the REST API is secured using HTTPS to prevent replay and injection attacks. We assume that the IoT application already uses TLS for other services. Therefore, it has to pass the certificate to IoTWall. This avoids the IoT device having to store and administrators having to manage two certificates. Moreover, IoTWall uses the standardized HTTP Basic authentication scheme together with HTTPS to protect the REST API from unauthorized access, with known default credentials that can be changed via the API. To prevent attacks after temporary power loss, IoTWall persists all rules in the EEPROM. Hence, IoTWall automatically loads the latest ruleset from the EEPROM at each start of the IoT device.

IV. Evaluation

IoTWall targets resource-constrained IoT devices where low memory utilization and power consumption are crucial. In this section, we measure power consumption, memory utilization, latency, and throughput of an ESP32 to evaluate if IoTWall reaches the design goals **G3** and **G4** (small size and computational efficiency).

A. Evaluation Testbed

For the evaluation, we set up a testbed with an example IoT application that uses the IoTWall library. Figure 2 shows the testbed components to send packets to the firewall and to measure power consumption and latency. The packet generator sends IPv4 UDP packets to the ESP32 via a wireless network. The IoT application is a UDP server that listens for incoming packets and returns them to their origin IP address. In between, a switch inserts timestamps into packets. We use the difference between the first and the second timestamp in each packet to calculate the latency. Hence, the latency includes the wireless connection and the packet processing of the ESP32. For reproducibility, we repeat the measurement 100 times and use the wireless network exclusively for the ESP32.

The ESP32 offers power management features to save energy in idle situations. We disable these features to not measure the wakeup routine and frequency changes but the actual IoTWall operation. This keeps the CPU frequency constant at the cost of higher power consumption. We configure a CPU frequency of $160\,\mathrm{MHz}$, which is one of four configurable frequencies between $40\,\mathrm{MHz}$ and $240\,\mathrm{MHz}$. It provides a balance between power consumption and speed. With power management features enabled, the power consumption and latency will be different but the change of those between different configurations will be similar.

For the measurement traffic, we used a packet size of $128\,\mathrm{B}$, assuming IoT devices send or receive mostly a small amount of data. Additionally, small packets have fewer collisions on the wireless channel, so our evaluations have a higher reproducibility. In order to analyze the impact of IoTWall, we vary the rule count between 0 and 100 as well as the data rate between $0.1\,\mathrm{Mbit/s}$ and $1\,\mathrm{Mbit/s}$. We configure at most 100 rules, assuming that far fewer than 100 rules are required to secure an IoT device in the field. Additionally, we compare all measurements to the same IoT application without the IoTWall library included.

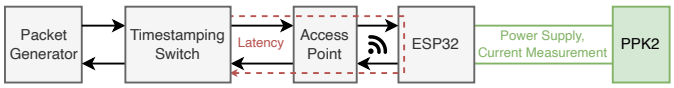


Fig. 2. Testbed topology for the evaluation

B. Power Consumption

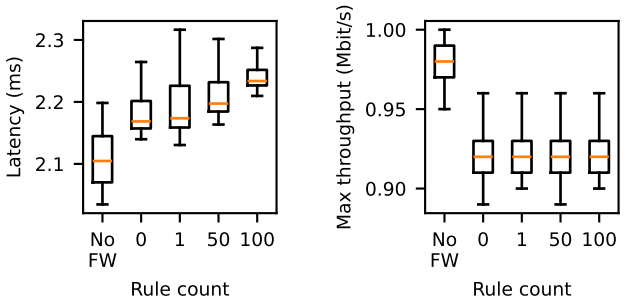
To achieve a battery life of multiple years, the power consumption of an IoT device must be low. We determine the power consumption of the ESP32 by measuring its current draw under a constant voltage. To automate the current measurement, we use the *Nordic Power Profiler Kit II (PPK2)*. It has a resolution of $1\,\mathrm{mA}$, an accuracy of $15\,\%$, and an offset of $5\,\%$ for the measurement range of $50\,\mathrm{mA}$ to $1000\,\mathrm{mA}$ [8].

We measured the current draw of the ESP32 with different data rates and numbers of rules. At a data rate of $0.1\,\mathrm{Mbit/s}$, the mean current draw is $114\,\mathrm{mA}$, with almost no change between different numbers of rules. Higher data rates lead to an increase in current draw, e.g., at $0.8\,\mathrm{Mbit/s}$ the current draw is $130\,\mathrm{mA}$ without IoTWall and $130.5\,\mathrm{mA}$ with 100 rules. All measurement results are within the accuracy of PPK2. Still, repeating the measurements 100 times shows a clear picture: The data rate impacts the current draw, whereas the number of rules only increases the current draw by less than $1\,\mathrm{mA}$. Compared to these measurements under load, the current draw of the ESP32 with IoTWall but without packet transmission is $57\,\mathrm{mA}$, which is half of the current draw under load. We conclude that the firewall does not increase power consumption significantly, as the data rate has a much larger impact on power consumption.

C. Application and Configuration Size

The application binary and the data must be stored on the IoT device. In case of IoTWall, the data encompasses the configuration and the ruleset, which must be persisted between restarts. There are two types of memory for this: flash memory for the application and EEPROM for data. Since EEPROM is expensive, many microcontrollers, such as the ESP32, emulate the EEPROM using flash memory to save costs. If microcontrollers use an EEPROM, it is typically small. The ESP32 we use has $4\,\mathrm{MB}$ of flash memory. However, the flash memory must store the application, data, and provide space for over-the-air updates. Other chips have less flash memory, e.g., the STM32L4 series includes $64\,\mathrm{kB}$ to $1024\,\mathrm{kB}$ of flash [9]. As a result, IoTWall should only require a small amount of flash memory for the application and data.

To determine the increase in size of the application binary, we compile the example application with and without IoTWall and compare the size. For this, we use the command-line tool `idf.py` that is supplied with the ESP-IDF toolchain. Calculating the difference between the two *total image sizes* indicates that IoTWall requires only about $19\,\mathrm{kB}$ of additional



(a) Latency at $0.1\,\mathrm{Mbit/s}$ (b) Maximum throughput

Fig. 3. Measurements at different rule counts (packet size: 128 B)

flash memory. The REST API makes up about $15\,\mathrm{kB}$ of this amount, the filtering about $4\,\mathrm{kB}$. The firewall by Maheshwari et al. [3] requires only $316\,\mathrm{B}$ of flash memory. However, they do not describe its features and flexibility, making it impossible to compare with IoTWall. The flash size of $4\,\mathrm{MB}$ on the ESP32 is much larger, which means that the size of IoTWall is negligible. In case of a microcontroller with less flash memory, the impact of IoTWall depends on the available flash memory.

The memory needed for the configuration and ruleset comprises a fixed and a variable component. Configuration parameters, e.g., the API password, use a fixed amount of memory, whereas the ruleset depends on the number of rules. In sum, IoTWall requires $67\,\mathrm{B}$ for the fixed-size configuration, $20\,\mathrm{B}$ per IPv4 rule, and $32\,\mathrm{B}$ per IPv6 rule. Even with 100 rules, which is rather unrealistic, the configuration size is small compared to the flash memory of $4\,\mathrm{MB}$ on the ESP32.

D. Latency and Throughput

With IoTWall, the IoT device has to perform additional processing for every packet due to packet filtering. As a result, packets take longer to process, leading to a higher latency and reduced maximum throughput. Hence, we analyze the impact of IoTWall on the packet latency and the maximum throughput of the IoT device. Typically, IoT devices are connected through a wireless network, just as the ESP32 in our testbed. The latencies we measure include the latency and the jitter of the wireless network. This means that the latencies measured on the testbed are comparable to the latencies of deployed IoT devices. Consequently, the absolute numbers and the observed jitter are larger than what the processing of the IoT device introduces. However, the changes between the measurements still indicate the impact of IoTWall.

Figure 3a shows the latency at a data rate of $0.1\,\mathrm{Mbit/s}$ with varying numbers of rules with and without the firewall. The boxplots show the distribution of the median latency of each measurement, with the whiskers marking the minimum and maximum median latency. The measurements show that with an increasing number of rules, the median latency increases slightly from $2.1\,\mathrm{ms}$ without IoTWall to $2.2\,\mathrm{ms}$ with 100 rules. However, compared to the overall latency and jitter, the latency difference between the different numbers of rules is small.

Figure 3b shows the maximum throughput with different numbers of rules with and without the firewall. We measure the maximum throughput by increasing the data rate until packet loss occurs. Due to the wireless transmission in the testbed, we observe packet loss at every data rate. However, packet loss suddenly increases at the maximum data rate, as the ESP32 becomes overloaded. Hence, we define that the maximum throughput is reached when the packet loss exceeds $1\,\%$. The results show a median throughput of $0.98\,\mathrm{Mbit/s}$ without IoTWall. With IoTWall, the throughput decreases slightly to $0.92\,\mathrm{Mbit/s}$, as IoTWall must parse the IP and transport headers and then must iterate over the ruleset to check the rules. Increasing the rule count to 100 does not change the median throughput. This means that the additional processing, i.e., comparing the packet headers with each rule, has only a small impact on the processing of a packet. Typically, IoT devices do not need to process packets at a high rate. For example, they only send sensor readings every few seconds or minutes. Therefore, we conclude that a small decrease in throughput is acceptable for IoT devices.

V. Conclusion

In this paper, we introduced *IoTWall*, a lightweight host firewall library for IoT devices. Developers can integrate IoTWall in existing IoT applications for ESP32 microcontrollers with few additions to the application code. We defined four design goals that aim to allow an easy integration into existing IoT applications, provide easy configuration of the ruleset at scale, and to keep the resource utilization low. Based on these design goals, we present the design of IoTWall. We measured power consumption, memory utilization, latency, and throughput to analyze the impact of IoTWall with regard to the design goals on an ESP32. The measurements show a small degradation in all areas, but still in an acceptable range. By publishing the IoTWall implementation, we encourage developers to improve the security of IoT devices by integrating a host firewall.

References

[1] D. Ziegelmeier and S. Goldschmidt. “lwIP,” Accessed: Mar. 11, 2025. [Online]. Available: https://savannah.nongnu.org/projects/lwip/.

[2] M.-O. Pahl, F.-X. Aubet, and S. Liebald, “Graph-Based IoT Microservice Security,” in *IEEE/IFIP Network Operations and Management Symposium*, 2018.

[3] N. Maheshwari and H. Dagale, “Secure Communication and Firewall Architecture for IoT Applications,” in *International Conference on Communication Systems & Networks (COMSNETS)*, 2018.

[4] J. McManus. “ESP8266-Firewall,” Accessed: Mar. 11, 2025. [Online]. Available: https://github.com/joemcmanus/ESP8266-Firewall.

[5] wolfSSL Inc. “wolfSentry,” Accessed: Mar. 11, 2025. [Online]. Available: https://github.com/wolfSSL/wolfsentry.

[6] Check Point Software Technologies Ltd. “IoT Device Security for Manufacturers,” Accessed: Mar. 11, 2025. [Online]. Available: https://www.checkpoint.com/quantum/iot-protect/iot-device-security/.

[7] Sectigo Limited. “Sectigo IoT Security & Identity Management Advancements Speed Integration and Use in Multi-Vendor Ecosystems,” Accessed: Feb. 13, 2025. [Online]. Available: https://www.sectigo.com/resource-library/sectigo-iot-security-identity-management-advancements-speed-integration-and-use-in-multi-vendor-ecosystems.

[8] Nordic Semiconductor ASA. “Measurement Accuracy,” Accessed: Mar. 18, 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ug_ppk2/page/UG/ppk/ppk_measure_accuracy.html.

[9] STMicroelectronics. “STM32L4 series,” Accessed: Mar. 20, 2025. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html.