Deep En-Route Filtering of Constrained Application Protocol (CoAP) Messages on 6LoWPAN Border Routers

Felix Seidel
Hasso Plattner Institute
University of Potsdam, Germany
Felix.Seidel@student.hpi.de

Konrad-Felix Krentz
Hasso Plattner Institute
University of Potsdam, Germany
Konrad-Felix.Krentz@hpi.de

Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany
Christoph.Meinel@hpi.de

***Abstract*—Devices on the Internet of Things (IoT) are usually battery-powered and have limited resources. Hence, energy-efficient and lightweight protocols were designed for IoT devices, such as the popular Constrained Application Protocol (CoAP). Yet, CoAP itself does not include any defenses against denial-of-sleep attacks, which are attacks that aim at depriving victim devices of entering low-power sleep modes. For example, a denial-of-sleep attack against an IoT device that runs a CoAP server is to send plenty of CoAP messages to it, thereby forcing the IoT device to expend energy for receiving and processing these CoAP messages. All current security solutions for CoAP, namely Datagram Transport Layer Security (DTLS), IPsec, and OSCORE, fail to prevent such attacks. To fill this gap, Seitz et al. proposed a method for filtering out inauthentic and replayed CoAP messages "en-route" on 6LoWPAN border routers. In this paper, we expand on Seitz et al.'s proposal in two ways. First, we revise Seitz et al.'s software architecture so that 6LoWPAN border routers can not only check the authenticity and freshness of CoAP messages, but can also perform a wide range of further checks. Second, we propose a couple of such further checks, which, as compared to Seitz et al.'s original checks, more reliably protect IoT devices that run CoAP servers from remote denial-of-sleep attacks, as well as from remote exploits. We prototyped our solution and successfully tested its compatibility with Contiki-NG's CoAP implementation.**

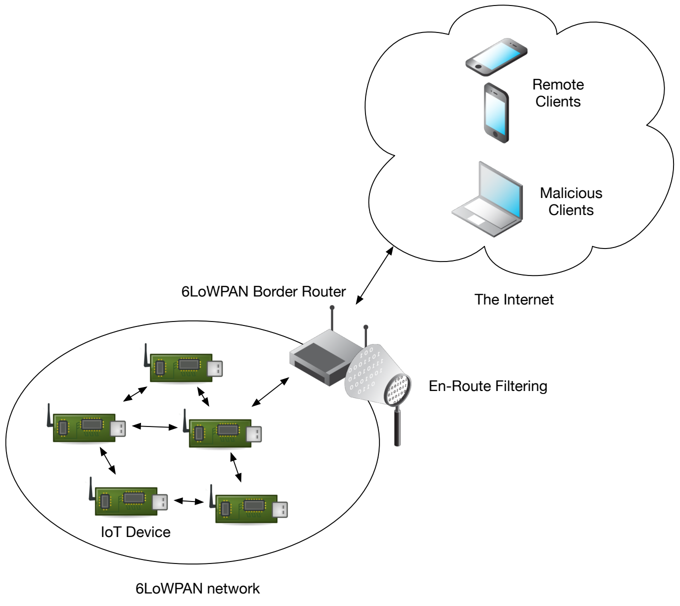

Fig. 1. Deployment Scenario

I. Introduction

An increasing number of devices participate in the IoT, most of which have low processing power, small amounts of memory, and limited communication bandwidth. Moreover, IoT devices are often battery powered and hence must operate energy efficiently. In view of these challenges, the Internet Engineering Task Force (IETF) proposed a whole new protocol stack, which builds upon the IEEE 802.15.4 radio standard [1]. Directly on top of IEEE 802.15.4, the 6LoWPAN adaption layer conveys IPv6 packets over IEEE 802.15.4 links [2]. Thereupon, the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) is recommended for routing IPv6 packets through 6LoWPAN networks [3]. Finally, CoAP is an energy-efficient replacement for HTTP [4].

Yet, in addition to more traditional attack vectors, such as remote exploits, battery-powered IoT devices are also vulnerable to so-called denial-of-sleep attacks [5]. Denial-of-sleep attacks generally deprive victim devices of entering low-power sleep modes, thereby draining their charge. Repercussions of denial-of-sleep attacks include long outages, violated quality of service guarantees, and a reduced customer satisfaction. Therefore, it is highly desirable to protect IoT devices against denial-of-sleep attacks. Nevertheless, the state of the art is that Internet-located attackers can force IoT devices that run CoAP servers to expend much energy by sending lots of CoAP messages to them. Also, recommended security solutions for CoAP, namely DTLS [6], IPsec [7], and OSCORE [8], provide no protection against such denial-of-sleep attacks since these security solutions filter out injected CoAP messages only after an IoT device expended energy for receiving them already.

As a countermeasure Seitz et al. proposed to filter out inauthentic CoAP messages "en-route", i.e., before they get into a 6LoWPAN network. Their basic idea is to add filtering capabilities to the 6LoWPAN border router, as shown in

978-1-5386-4980-0/19/$31.00 © 2019 IEEE

Figure 1. Usually, the 6LoWPAN border router operates at the network layer and only takes care of forwarding IPv6 packets. In addition, Seitz et al. provided the 6LoWPAN border router with the capability to filter out inauthentic and replayed CoAP messages. For this, they assume that CoAP clients authenticate and encrypt each CoAP message at the application layer using a symmetric key that is shared with both the target CoAP server and the 6LoWPAN border router.

While Seitz et al.'s countermeasure prevents denial-of-sleep attacks against CoAP servers, their solution depends on the secrecy of all symmetric keys. That is, as soon as any symmetric key leaks, denial-of-sleep attacks become possible again. Moreover, as soon as any symmetric key leaks, Internet-located attackers may also try to exploit software vulnerabilities of CoAP servers so as to penetrate into a 6LoWPAN network.

In this paper, we make two main contributions: First, we revise the software architecture of Seitz et al. to enable the implementation of a wider range of filtering rules. Second, we propose new filtering rules, which mitigate the weaknesses of Seitz et al.'s original design. In fact, our new filtering rules keep up the protection against denial-of-sleep attacks even if any number of symmetric keys leaks. Beyond that, our new filtering rules complicate remote exploits since they ensure the validity of CoAP messages, too.

This paper is organized as follows. Section 2 introduces related work. Section 3 presents our software architecture, as well as our new filtering rules. Section 4 details our prototypical implementation. Section 6 concludes and discusses areas of future work.

II. Related Work

Amin et al. identified three kinds of attacks on 6LoWPAN networks, namely attacks from hosts on the Internet, attacks from within the network on other nodes, and attacks from nodes within the network on Internet clients [9]. Much prior work has been done on mitigating attacks from within the network. For example, a typical attack from within the network is a Path-based Denial-of-Service (PDoS) attack [10]. In a PDoS attack, a malicious node sends packets to distant destinations, thereby causing each node along the path to expend energy for receiving, processing, and forwarding. However, for the scope of this paper, our focus is on attacks from hosts on the Internet.

To mitigate attacks from hosts on the Internet, Zhang et al. describe four different authentication models, namely authentication by (i) gateway, (ii) security token, (iii) trust chain, and (iv) global trust tree [11]. While these four models are applicable to securing 6LoWPAN networks against attacks from the Internet, not all models do mitigate denial-of-sleep attacks against CoAP. This is because authentication by security token, trust chain or global trust tree implement end-to-end security, i.e., verify the authenticity of messages on the IoT devices, as is also done in DTLS, IPsec, and OSCORE. By contrast, in the authentication by gateway model, messages are filtered out before they reach an IoT device. Although Zhang et al. argue that the gateway could become a performance bottleneck, as for a 6LoWPAN network, a high number of devices in a network can be connected to the Internet with a single border router, which makes it economically feasible to use more powerful hardware for the border router.

A number of protocols were proposed for securing CoAP in particular. For example, IPsec [7] could be used to secure CoAP messages at the Internet Protocol (IP) layer. In recent years, protocols tailored for constrained devices were developed in order to mitigate common drawbacks of IPsec in an IoT context, e.g. high complexity and overhead. Besides, the CoAP standard recommends using the DTLS protocol [12]. While DTLS can be used to secure arbitrary protocols at the transport layer, the Object Security for Constrained RESTful Environments (OSCORE) protocol was developed in order to address a number of shortcomings of using DTLS together with CoAP, such as end-to-end security with proxies. Although all of those protocols are capable of ensuring information confidentiality and integrity, neither of those can be used to prevent denial-of-sleep attacks because an IoT device still needs to expend energy for receiving and processing injected CoAP messages.

Seitz et al. presented a concrete solution for en-route filtering of CoAP messages on a trusted border router in order to mitigate denial-of-sleep attacks [13]. Their approach works as follows. The client, border router, and IoT devices share a Pre-shared Key (PSK). When the client sends a CoAP message to a device, it encrypts the message payload with the PSK and adds a Keyed-Hash Message Authentication Code (HMAC) as a custom CoAP option. Upon receipt of the CoAP message, the border router decrypts the CoAP message, calculates the valid HMAC value for the message's payload, and compares it to the HMAC in the CoAP message. If both values match, the message is forwarded; otherwise, it is discarded. The solution also involves the use of counters to prevent replay attacks. Their en-route filter implementation is integrated into the embedded border router of Contiki, which is a lightweight operating system for IoT devices.

However, Seitz et al.'s solution depends on the secrecy of PSKs, as mentioned in the introduction. This is highly critical because attackers may get access to such keys, e.g., via side-channel attacks. Subsequently, denial-of-sleep attacks become possible again. Moreover, once a PSK leaks, attackers may also send specially-crafted CoAP messages so as to remotely exploit software vulnerabilities of IoT devices.

In the next section, we will address these limitations of Seitz et al.'s solution.

III. Integration Method & Filtering Rules

The main building block of our solution is an extensible software architecture for filtering CoAP messages en-route before they enter the IoT network. Our software architecture fulfills the following three high-level requirements. First, our software architecture allows the user to implement arbitrarily complex logic as to how the CoAP messages are processed and filtered (*flexibility*). Second, our en-route filter is easy to extend

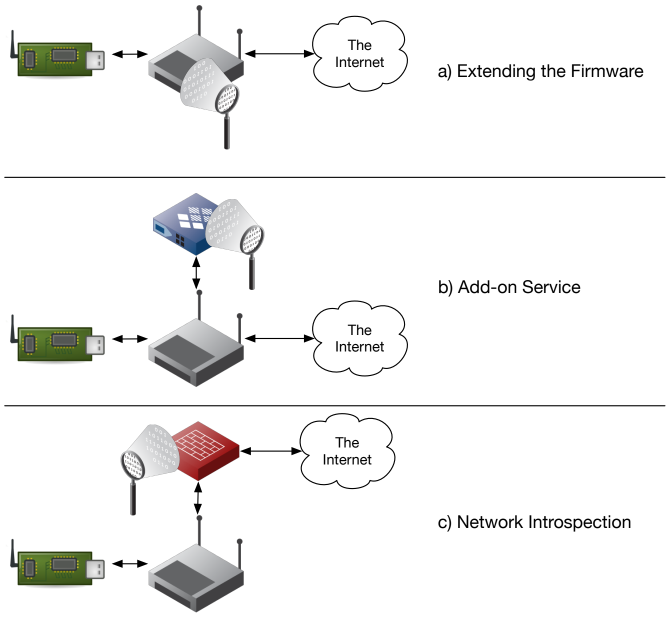

Fig. 2. Methods for filtering CoAP messages at the border router

and customize (*extensibility*). That is, implementing additional processing rules does not require in-depth knowledge on how the message filtering is implemented. Furthermore, existing filtering rules are customizable by the end user. Third, the en-route filter is easy to set up and use in the field (*ease of implementation*).

A. En-Route Filter Integration Method

The first design decision was to choose where to integrate the en-route filter in the border router software stack. We considered three possible integration methods. In the following, we will describe each of these options and evaluate them regarding flexibility, extensibility, and ease of implementation.

The first option is to add the packet filtering logic to the border router firmware, as shown in Figure 2a. This option expands on how embedded operating systems implement 6LoWPAN border routers at the moment and is also taken by Seitz et al. For example, Contiki-NG provides two modes of running a 6LoWPAN border router, namely the embedded and the native mode. In the embedded mode, an IoT device acts as a 6LoWPAN border router. Furthermore, this IoT device is either directly connected with an external network via Ethernet or indirectly via a PC that, on the one hand, communicates with an external network and, on the other hand, with the IoT device via USB. Alternatively, in the native mode, a PC also relays IPv6 packets between an external network and a USB-connected IoT device, but the PC takes over the task of running upper-layer protocols, such as RPL [3]. That is, in the native mode, some processing tasks are moved from the IoT device to the PC.

As for the embedded mode, integrating the en-route filter into the firmware of the IoT device that runs the 6LoWPAN border router seems straightforward, resulting in a high *ease of implementation*. Yet, in this case, the en-route filter would have to be implemented in the same programming language as was used to create the firmware. Typically, this firmware is written in C and hence the en-route filter would have to be implemented in C, too. This conflicts with our requirement that maintaining and adding new filtering rules should be easy for the end user. Moreover, since the en-route filter would run on an IoT device, all filtering rules would need to cope with the limited resources of that IoT device.

As for the native mode, the en-route filter could be integrated into the software that runs on the PC. While this would, at least, resolve the resource limitations, it would still be necessary to implement the filtering rules in the C programming language since, as far as Contiki-NG is concerned, the PC software is written in C, as well.

The second option is to extend the border router firmware by adding code to relay incoming packets to an add-on service for filtering, as shown in Figure 2 b. The add-on service would communicate with the border router firmware over a network-based Application Programming Interface (API). Therefore, the add-on service could be implemented in an arbitrary programming language as long as it supports the API to communicate with the border router. This would allow the end-user to implement filtering rules in a more user-friendly programming language such as Python, resulting in higher *flexibility* and *extensibility*. On the other hand, setting up and running the add-on service would increase the complexity of the solution. Thus, we consider the *ease of implementation* of this option to be lower than that of the first option because of the additional complexity of installing and configuring such an add-on service.

The third option, which we call network introspection, is to implement the en-route filter independent of the border router firmware. With this option, we would process incoming messages from the Internet before routing the messages to the border router. However, the en-route filter must be run on a separate PC (like in the case of running Contiki's border router in native mode), potentially leading to a more complex network architecture and higher hardware costs. On the other hand, the en-route filter could be used in conjunction with any border router firmware because the filtering is transparent to the border router. Further, since the network introspection method is independent of any border router, it could also be reused in other use cases than filtering CoAP messages, such as for filtering ICMPv6 messages. Overall, this method combines the *flexibility* and *extensibility* of an add-on service with a high *ease of implementation*.

According to our discussion, network introspection seems most promising in terms of flexibility, extensibility, and ease of implementations. Consequently, we opted for this third option, deviating from Seitz et al.

B. CoAP Message Filtering Rules

In order to validate the extensibility of our architecture, we implemented examples of CoAP message filtering rules. In the following, we will describe how each of those rules operates.

Then, we will introduce a set of properties of CoAP message filtering rules according to which we characterize our filtering rules.

First, we implemented the message authenticity verification method by Seitz et al. Our rule is compatible with their implementation in the border router firmware on the CoAP protocol level, but it does not require the IoT devices to verify the authenticity a second time. Instead, it decrypts the messages on the fly and forwards the unencrypted message to the IoT device.

The second rule applies rate limiting to CoAP messages. It uses leaky bucket counters to limit the number of messages to a device per remote client IP address. A leaky bucket counter is "a counter that (a) is incremented by unity each time an event occurs and (b) is periodically decremented by a fixed value" [14]. A separate Leaky Bucket Counter is used for each distinct pair of source and destination IP address. Each bucket is configured with a bucket capacity (i.e., the maximum number of requests in a time period) and a leak rate (i.e., the time period after which the bucket is empty). For example, we can limit a remote client to 10 requests per IoT device and minute. For each incoming message, we check if the bucket capacity is depleted, and subsequently drop the message if this is the case. See Figure 3 for an example of how the rate limiting rule uses Leaky Bucket Counters for two remote clients and a single IoT device.

Third, we implemented a CoAP resource filtering rule. Its purpose is to reject CoAP requests to non-existent resources on an IoT device. The CoAP resource option is set in the path portion of a CoAP Uniform Resource Identifier (URI). To achieve this, we use the Constrained RESTful Environments (CoRE) Link format as described in [15]. A device can publish resource discovery information at a standardized resource `/.well-known/core`. See Figure 4 for an example response which defines two resources: `/.well-known/core`

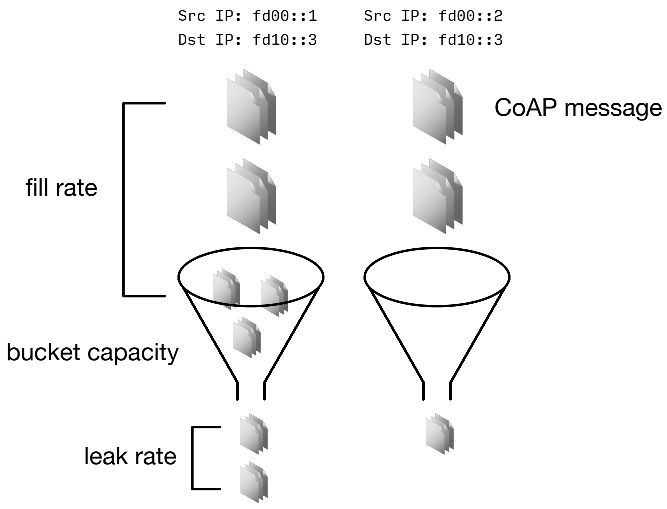

Fig. 3. Example of using leaky bucket counters for rate limiting of CoAP messages

```
</.well-known/core>;ct=40,
</actuators/leds>;title="LEDs,
POST/PUT";rt="Control"
```

Fig. 4. Resource Discovery Information in the CoRE link format

TABLE I
PROPERTIES OF COAP MESSAGE FILTERING RULES

	Rule Name	Statefulness	Depth	Action
a)	Rate Limiting	Stateful	Metadata	Filter
b)	Resource Filtering	Stateless	Metadata	Filter
c)	Authentication	Stateless	Payload	Modify

and `/actuators/leds`. Our resource filtering rule retrieves this description for each IoT device and compares the requested resource with the resources found in the resource discovery information. Consequently, messages to non-existent resources are dropped.

When we implemented our CoAP message filtering rules, we found three distinct properties of filtering rules, as shown in Table I.

First, a rule can be either stateful or stateless, describing whether the result of the rule is dependent on the history of messages. Stateful rules need to store information on the packets they have filtered. An example of a stateful rule is rate limiting.

Next, a rule can be metadata-oriented or payload-oriented. Metadata-oriented rules need only packet metadata (such as IP or CoAP headers) as input while payload-oriented rules process the payload, too. An example of a metadata-oriented rule is checking the source IP address of a packet or the resource in a CoAP request. Metadata-oriented rules typically need fewer resources for processing than payload-oriented rules. The authenticity verification rule is an example of a payload-oriented rule since the border router calculates an HMAC of the request payload to verify authenticity.

Last, rules can be filtering or modifying. The outcome of a filtering rule is a decision whether the processed message should be forwarded while the outcome of a modifying rule is one or more CoAP messages that may differ from the original message. For example, the authenticity verification transparently decrypts messages before forwarding them to the IoT device.

Our software architecture supports rules with all described properties in order to satisfy the *flexibility* requirement.

IV. IMPLEMENTATION

We implemented a prototype of an en-route filter, as well as our filtering rules following the network introspection method. We chose Linux as the target operating system for our implementation because of its high popularity for routers and servers. Furthermore, the packet filtering framework in the Linux kernel (*Netfilter* [16]) already includes a module with a kernel API for intercepting and filtering network packets in userspace. The actual filtering rules are written in *Golang* [17].

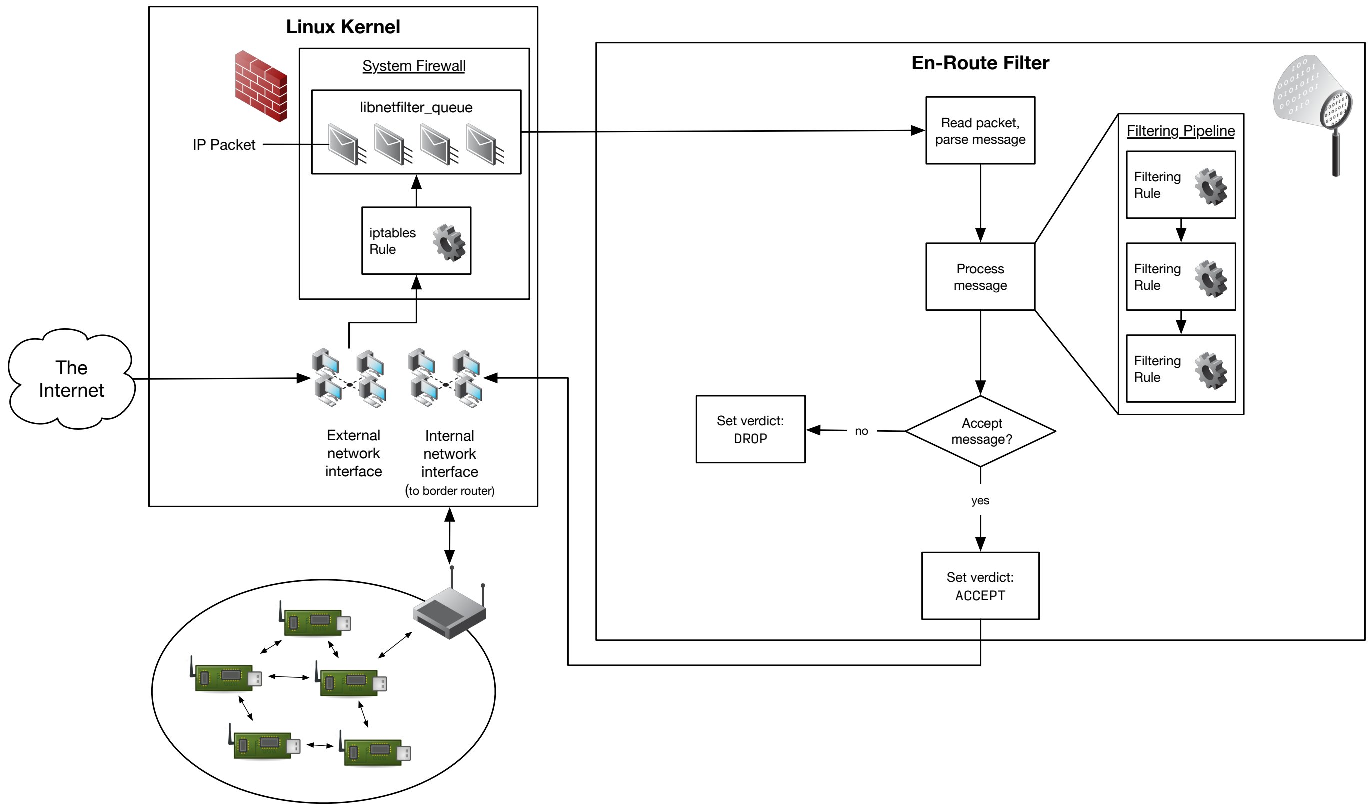

Fig. 5. Flowchart of the en-route filtering process

The process for intercepting and processing incoming CoAP messages works as shown in Figure 5. Essentially, IP packets are read from a queue, message filtering rules are applied, and the message is forwarded to the IoT device if it is accepted.

In order to introspect incoming network traffic, we added a rule to the Linux system firewall, which passes all messages into a queue. For this purpose, we use Netfilter's *libnetfilter_queue* module [18]. This allows us to retrieve intercepted packets in a userspace program and issue a verdict whether the packet should be accepted or dropped. The packets read from the queue are represented as a byte stream which contains the OSI layers 3 (Network Layer) to 7 (Application Layer). Next, the byte stream is decoded, the individual layers are decomposed, and the message is parsed.

In order to decide whether the message should be forwarded, we apply filtering rules in the *filtering pipeline*. Each filtering rule in the pipeline accepts a message as its input, potentially modifies the message, and returns a verdict (i.e., `ACCEPT` or `DROP`) as its result. The filtering pipeline is reused for each processed message to allow each rule to persist data on processed messages for implementing stateful filtering rules. The message is discarded if any filtering rule returns `DROP` as its verdict. If all rules in the filtering pipeline return `ACCEPT` as their verdict, we set the verdict of the original packet in the *libnetfilter* queue to `ACCEPT` so that the kernel forwards the packet to the IoT device. Otherwise, we set the verdict to `DROP` so that the packet is discarded.

V. Conclusions & Future Work

As low-power and low-resource connected devices rapidly become more and more popular in both industrial and home applications, the importance of protecting them from attacks from the Internet when designing such systems increases as well. With our network introspection method, we provide a generic and flexible way to implement filtering rules in a wide range of scenarios, such as for filtering out CoAP messages en-route, but also for filtering out ICMPv6 and other traffic en-route. Our examples of deep filtering rules provide a starting point for deploying a comprehensive filtering framework to border routers. Currently, our filtering rules already perform basic protocol validation in order to complicate remote exploits, as well as authentication, replay protection, and rate limiting so as to prevent remote denial-of-sleep attacks.

We suggest two avenues for future research. First, since CoAP handles fragmentation of packets at the application layer, a stateful filtering rule for block-wise transfers [19] could be implemented so that the entire payload of a fragmented CoAP message can be processed.

Second, another useful feature is supporting multiple border routers in larger 6LoWPAN networks. This presents a challenge for stateful filtering rules which would need to share their state between each 6LoWPAN border router.

References

[1] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized protocol stack for the internet of (important) things," *IEEE communications surveys & tutorials*, vol. 15, no. 3, pp. 1389–1406, 2013.

[2] J. Hui and P. Thubert, "Compression format for ipv6 datagrams over ieee 802.15.4-based networks," Internet Requests for Comments, RFC Editor, RFC 6282, September 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6282.txt

[3] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "Rpl: Ipv6 routing protocol for low-power and lossy networks," Internet Requests for Comments, RFC Editor, RFC 6550, March 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6550.txt

[4] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," Internet Requests for Comments, RFC Editor, RFC 7252, June 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7252.txt

[5] M. Brownfield, Y. Gupta, and N. Davis, "Wireless sensor network denial of sleep attack," in *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*. IEEE, 2005, pp. 356–364.

[6] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of ipv6 packets over ieee 802.15.4 networks," Internet Requests for Comments, RFC Editor, RFC 4944, September 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4944.txt

[7] S. Frankel and S. Krishnan, "Ip security (ipsec) and internet key exchange (ike) document roadmap," Internet Requests for Comments, RFC Editor, RFC 6071, February 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6071.txt

[8] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, "Object security for constrained restful environments (oscore)," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-core-object-security-13, June 2018. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-core-object-security-13.txt

[9] S. O. Amin, Y. jig Yoon, M. S. Siddiqui, and C. S. Hong, "A novel intrusion detection framework for ip-based sensor networks," in *Information Networking, 2009. ICOIN 2009. International Conference on*. IEEE, 2009, pp. 1–3.

[10] J. Deng, R. Han, and S. Mishra, "Defending against path-based dos attacks in wireless sensor networks," in *Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks*. ACM, 2005, pp. 89–96.

[11] Z.-K. Zhang, M. C. Y. Cho, and S. Shieh, "Emerging security threats and countermeasures in iot," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 1–6.

[12] E. Rescorla and N. Modadugu, "Datagram transport layer security," Internet Requests for Comments, RFC Editor, RFC 4347, April 2006. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4347.txt

[13] K. Seitz, S. Serth, K.-F. Krentz, and C. Meinel, "Enabling en-route filtering for end-to-end encrypted coap messages," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017, p. 33.

[14] M. Weik, *Computer science and communications dictionary*. Springer Science & Business Media, 2000.

[15] Z. Shelby, "Constrained restful environments (core) link format," Internet Requests for Comments, RFC Editor, RFC 6690, August 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6690.txt

[16] H. Welte and P. N. Ayuso. (2014) netfilter project homepage. [Online]. Available: https://www.netfilter.org

[17] G. team. (2018) The go programming language. [Online]. Available: https://golang.org

[18] H. Welte and P. N. Ayuso. (2014) The netfilter.org "libnetfilter_queue" project. [Online]. Available: https://www.netfilter.org/projects/libnetfilter_queue/index.html

[19] C. Bormann and Z. Shelby, "Block-wise transfers in the constrained application protocol (coap)," Internet Requests for Comments, RFC Editor, RFC 7959, August 2016.