Arabian Journal for Science and Engineering (2020) 45:3123–3132
https://doi.org/10.1007/s13369-019-04297-5

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Information Flow Tracking and Auditing for the Internet of Things Using Software-Defined Networking

Bander Alzahrani[1]

Received: 10 September 2019 / Accepted: 6 December 2019 / Published online: 11 December 2019

Abstract
Internet of Things (IoT) applications are expected to have access to sensitive data. On the other hand, IoT devices are not powerful enough to implement complex security solutions. But even if this was not a problem, there are so many other security challenges related to the IoT that a security breach should not be considered an exceptional case. In this paper, we propose an information tracking mechanism that can be used as an audit tool in case of a security incidence. To achieve this goal, we leverage software-defined networking (SDN). SDN is a promising technology that receives increasing attention, since it enables network "programmability" and intelligent packet forwarding. In this paper, we consider the case of an SDN deployment in the network of a single provider that interconnects various IoT devices. Furthermore, specialized network attachments points are used for parsing IoT specific protocols and perform the necessary actions. Moreover, in order to make our solution even more realistic and to facilitate deployment, we base our constructions and design solely on existing SDN standards.

Keywords Information Centric Network · Software-defined networking · Internet of Things · Constrained Application Protocol

1 Introduction

Nowadays, where the manufacturing cost of small, intelligent devices has become very low, the Internet of Things (IoT) is becoming a reality. The IoT envisions tiny, powerless devices, often with sensing and actuation capabilities, connected to the Internet. These IoT devices will be capable of exchanging data with other devices or web application and services. The IoT is expected to affect many aspects of our life, since some of the predominant IoT use cases are: healthcare, home automation, food supply, road safety, intelligent transportation, smart energy, manufacturing automation and many others that are closely related to our daily routine. With the IoT, the boundaries between cyber and real world become blurred. However, in addition to the plethora of new possibilities and opportunities, the IoT creates many security and privacy concerns: IoT devices are expecting to have access to sensitive and critical information and a security or privacy breach not only may expose personal information but it may also be life threatening (for a review on IoT security issues, interested readers are referred to [1]). On the other hand, securing IoT devices is hard and even in highly secured systems breaches are inevitable. The IoT possesses some unique security challenges due to the nature of the IoT devices (i.e., the Things):

- Things lack the necessary computational power to perform complex tasks such as public-key encryption.
- Things are susceptible to tampering, hence long-lived secret keys should not exist in a Thing.
- Things may also be physically exposed to attackers; for example, they may be located in an outdoor location unprotected.
- Things lack storage and memory; therefore, they cannot store complex access control policies or big user datasets.
- Things may not be always connected; hence, a security update may not always be possible.
- IoT use cases, assumes Things that will be used for a long period. For example, in a building management system, there might be sensors that are supposed to operate for the whole lifetime of the building.

✉ Bander Alzahrani
baalzahrani@kau.edu.sa

[1] IS department, College of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia

For those reasons, it is clear that a security breach in an IoT system should not be treated as an unexpected event; instead, special precautions should be considered, and in such case, it should be possible to determine what caused that incident.

In this paper, we propose a security solution for the IoT that operates in the network level and can be used for auditing security breaches. In particular, we leverage software-defined networking (SDN) [2] and information flow tracking, to provide an IoT security solution within the perimeter of the network of an ISP. As an IoT protocol, we consider the Constrained Application Protocol (CoAP) [3]. CoAP is an emerging and lightweight protocol that can be implemented directly on the IoT devices without requiring any intermediate service (as opposed for example to Message Queuing Telemetry Transport (MQTT)). Furthermore, as we discuss in Sect. 2.2, CoAP uses the same URI-based resource identification mechanism as HTTP, and it follows a similar messaging pattern; hence, we have selected CoAP as the underlay IoT protocol in order to implement a solution that can be easily extended for HTTP. The proposed solution allows CoAP service providers to "tag" sensitive information. These tags are propagated inside the provider's network and a logging service records all operations related to sensitive tags. In case of a security incidence, an end user will be able to audit the logs and find out what caused the breach.

The contributions of our work are the following:

- We design a lightweight content tagging solution and an algorithm that allows end users to tag sensitive information, as well as a mechanism that allows the incorporation of the tags inside SDN messages and their propagation in the network. Furthermore, the same mechanism removes the tags from the transmitted packets before delivery, making them transparent to client applications.
- We design a mechanism that allows server endpoints to "announce" in the network the tags used for tagging sensitive information. In return, the network logs the usage of sensitive information and it provides audit services that can be used by legitimate users to resolve security issues.
- We design a mechanism that prevents tag manipulation by unauthorized users.
- We extend SDN operations to include tags, facilitating this way information flow monitoring.
- We provide a reference implementation for CoAP servers, CoAP clients, as well as log analysis mechanisms. The implemented CoAP server is designed for constrained devices.
- Our solution is based on existing SDN standards, and it is deployable and extends the functionality of an existing SDN controller.

The reminder of this paper is organized as follows. In Sect. 2, we introduce SDN and the CoAP protocol. In Sect. 3, we discuss related work in this area. We give a high-level overview of our solution in Sect. 4 and in Sect. 5, we present its design. We evaluate our implementation in Sect. 6 and discuss our conclusion in Sect. 7.

2 Background

2.1 Software-Defined Networking

Software-Defined Networking (SDN) is a novel technology that allows the centralized control of programmable switches. SDN switches are controlled by a centralized entity, known as the "network controller" (or simply controller) using a protocol such as OpenFlow [4]. OpenFlow allows the definition of rules per network "flow:" Whenever a new packet arrives in a switch, the switch checks its "flow table" for a related processing rule; if such a rule exists it is applied, otherwise the packet is forwarded to the controller, which in return instructs the switch how to handle it. SDN enables application-aware network configuration and facilitates the development of novel services. For example, Jarschel et al. [5] leveraged SDN to optimize the performance of YouTube video streaming, Mekky et al. [6] implemented a firewall, a load balancer and a content-aware server selection mechanism using SDN, and Fotiou et al. [7] proposed an SDN-based architecture that facilitates the deployment of new IoT services.

Figure 1 illustrates an example of an SDN-based network topology. In this figure, there are three virtual switches (VS1, VS2 and VS3) fully connected (black lines). Furthermore, all switches are connected to an SDN controller (red dotted line). VS1 has a flow rule that says, "Packets with destination 10.0.0.2 will be forwarded to VS2." For this reason, when such a packet arrives, it is immediately forwarded to VS2, and then to its final destination (blue lines). On the contrary, VS3 is not configured with any rule; hence, when a packet arrives, it asks the controller where to forward it (green lines). The controller responds and eventually VS3 forwards the packet to the correct destination.

2.2 The Constrained Application Protocol

The Constrained Application Protocol (CoAP) can be considered as a lightweight version of the HTTP protocol, designed for the Internet of Things (IoT). Since the IoT will be composed of lightweight devices with limited processing capabilities, supporting a full-fledged version of HTTP is not a viable option. CoAP is a binary protocol, implemented over UDP, and allows a constrained device to act as "server" and offer a "resource." CoAP resources are identified using a URI (similar to HTTP URIs). CoAP URIs are prefixed with "CoAP://" and are composed of a host identifier part

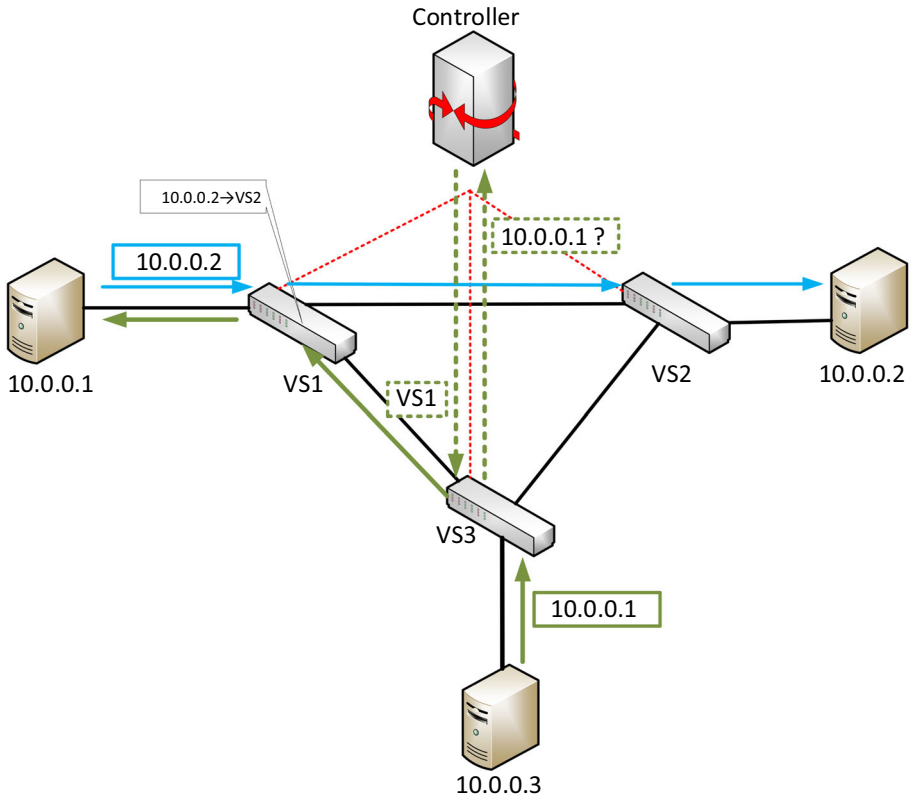

Fig. 1 An example of an SDN architecture

and a resource identifier part, separated by a slash ("/"). A CoAP client may issue a CoAP GET/PUT/POST request for a particular URI. Similar to HTTP, CoAP supports the use of proxies, that is, a CoAP client may send a CoAP request to a proxy and the proxy will handle it accordingly.

2.2.1 Messaging Model

The CoAP protocol specifies two kinds of messages: confirmable and non-confirmable. Confirmable messages must be acknowledged, whereas non-confirmable must not. All messages include a *message ID* so as to prevent duplicates. A message acknowledgement includes the same message ID as the message it acknowledges. There exists two cases of request–response flows:

1. A request is sent as a confirmable message and the server can respond immediately
2. A request is sent as a non-confirmable message or the server can respond immediately

In the first case, the server's response is "piggybacked" to the acknowledgement message (see section 5.2.1 of [3]). In the second case, we need a mechanism for matching a received response to a request. In this case, the request includes a *token* which is repeated in the response. In that case the response may be transmitted as a confirmable message (which needs to be acknowledged by the client). Figure 2 shows a message sequence example. In this figure, a CoAP client initially sends a confirmable request and the server responds immediately; therefore, the response is included in the acknowledgement message. After some time, the client sends a confirmable request but the server cannot respond immediately. Therefore, the server sends an empty acknowledgement and after some time it sends the response. Since the response is sent as a confirmable message the client sends an acknowledgement.

2.2.2 CoAP Extensions

Various extensions enhance the CoAP protocol with additional functionalities. Two CoAP extensions are of interest to this work, namely CoAP group communication [8] and CoAP Observe [9]. The former extension allows a CoAP client to send a request simultaneously to a group of CoAP servers. The second extension allows a CoAP client to register a "permanent" interest in a specific CoAP resource and every time this resource changes value the CoAP server sends an update to all interested clients. In order for a client to stop receiving resource updates, he has to "deregister" his interest.

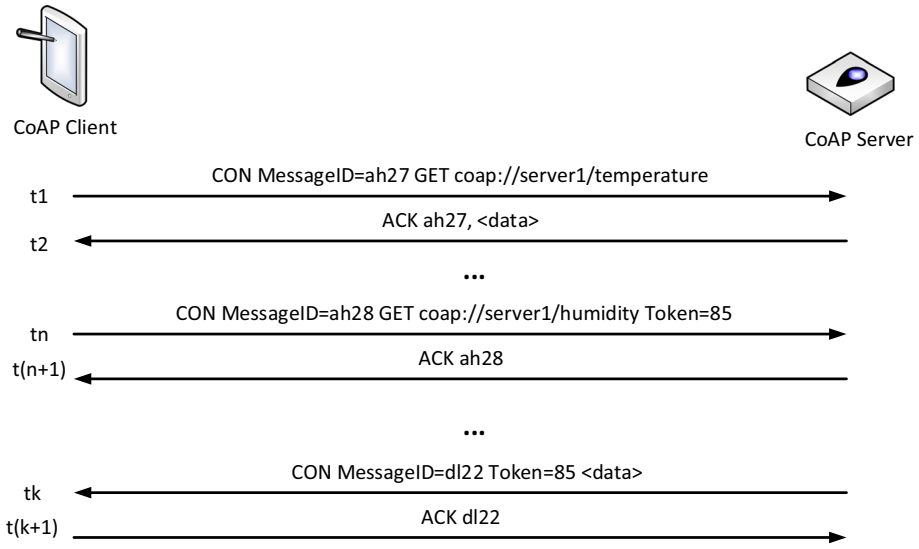

Fig. 2 CoAP messaging sequence. On the left part of the figure, the symbol tx represents the time

3 Related Work

3.1 Information Flow Tracking

Information flow tracking has been considered for similar purposes in other contexts.

3.1.1 Flow Tracking in Operating Systems

Information flow tracking has been widely studied in the context of operating systems. It has been used for analyzing malwares [10], for detecting zero-day attacks [11], for detecting unauthorized access to sensitive information [12], for detecting and analyzing software programming bugs [13], and in many other cases. A typical flow tracking mechanism (e.g., [14]) implements mechanisms that allow "tagging" of sensitive information directly in the computer memory. Then the tagged memory locations are continuously monitored: If a process modifies or moves the data stored in a monitored memory location, then this operation is recorded and, depending on the context, further actions are taken. Moreover, in case of (tagged) data transfer, the new location of the data is also monitored.

3.1.2 Flow Tracking in Mobile Devices

Mobile devices are another area where information flow tracking has received increased attention. TaintDroid [15] is an information flow tracking solution for the Android mobile operating system. TantDroid distinguishes between two types of applications: trusted and untrusted. A trusted application adds "tags" to sensitive data which are then monitored and managed the operating system's Java virtual machine. Untrusted applications are oblivious about the use of tags; hence, they cannot view them neither modify them. DroidSafe [16] has a similar goal. It achieves information flow monitoring by tagging data generated by "privacy sensitive" API calls (e.g., an API call that retrieves GPS measurements). All tags are then managed by the DroidSafe middleware in a transparent (for the applications) way.

3.1.3 Flow Tracking in Cloud Systems

Although the SDN technology has been used for implementing security solutions in Cloud systems, (e.g., Bawany et al. [17] develop a framework for detecting and mitigating DDoS attacks in large-scale networks built on SDN infrastructure, and Khaled et al. [18] build an overlay network that provides security services including intrusion detection systems, antivirus and DDoS prevention), flow tracking in a Cloud system is both a timely and hard problem. It is timely because more and more services are using shared infrastructure provided by Cloud providers.But it is hard since Cloud systems are multitenant environments and often one tenant is not aware of the rest. In particular, tracking and monitoring of IoT devices workload require efficient mechanisms for Cloud resource allocation (e.g., Al-Haidari et al. [19], leverage CPU scaling mechanisms to optimize Cloud resource allocation, whereas Calyam et al. [20] develop a tool that can capture various critical quality metrics and apply them in a resource allocation algorithm). CloudFence [21] tries to solve this problem by providing "flow tracking as a service." In particular, a Cloud tenant is allowed to tag sensitive data. Then a middleware, implemented by the Cloud provider, is responsible for monitoring the tagged data and record its usage: In case of a security incident, the data owner can audit the logs maintained by the Cloud provider and detect how and when sensitive data were accessed in an unauthorized way.

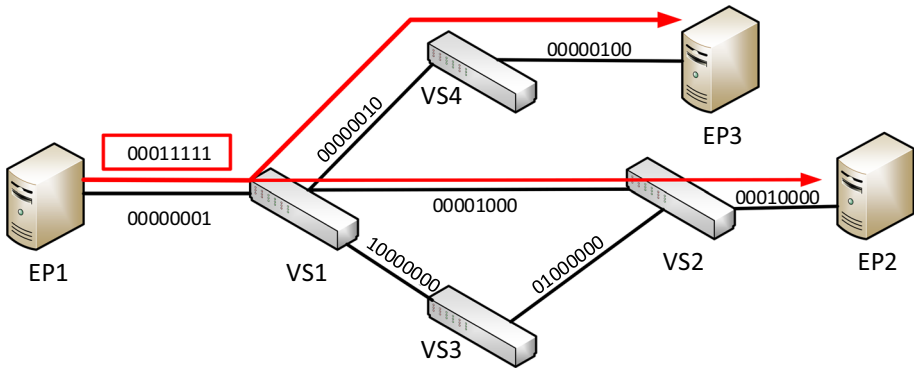

Fig. 3 Bloom filter-based forwarding

4 System Overview

4.1 Underlay Architecture

Our underlay architecture is based on the work published in [22]. The key characteristic of this architecture is that packet forwarding takes place using bloom filters stored in the IPv6 address fields (as proposed by Reed et al. [23]). Hence, all other packet fields can be used for defining OpenFlow rules. As we discuss in Sect. 6, we take advantage of this feature in order to store tags in the ethernet destination address field.

An simplified example of Bloom filter-based forwarding is illustrated in Fig. 3. In this type of forwarding, all links are identified by a unique binary array of fixed size. Whenever an entity wishes to send a packet to some endpoints, it constructs with the help of the SDN controller a Bloom filter by ORing the link identifiers of the links that compose the path from itself to the destination(s). For example, in Fig. 3 an endpoint, EP1 wants to send a packet to two other endpoints, EP2 and EP3; it creates a packet that includes in its header a Bloom filter (depicted in a red square); all intermediate switches examine the Bloom filter and decide in which links to forward it (this procedure is detailed in [23]). The work in [22] leverages this type of forwarding to decrease the network overhead when CoAP group communication and CoAP observe are used (i.e., the CoAP extension discussed in Sect. 2.2.2). In particular, CoAP group communication can be easily implemented without relying on IP-based multicast solutions. Similarly, when CoAP observe is used, resource updates are broadcasted to all interested CoAP clients, simultaneously, using a single packet.

In this work, we consider a single network provider (NP) that has deployed SDN-enabled switches in its core network and an SDN controller. The SDN network is managed using standard OpenFlow. Moreover, at the edges of this network, there exist *Network Attachment Points* (NAPs). In each NAP, there are devices connected to them (directly or indirectly), including IoT devices as well as more powerful nodes (e.g., PCs and Servers). For simplicity reasons, we assume that the NP provides only services related to our system.

4.2 System Entities

The main network entities of our architecture are CoAP endpoints. CoAP endpoints are distinguished to CoAP *clients* and *servers*. CoAP servers provide *resources* identified by well-known CoAP URIs. Similarly, CoAP clients perform requests for resources. A CoAP URI may be mapped to a single resource or to multiple resources (this is the case of CoAP group communication).

Our system extends NAPs with *CoAP proxies* that allow them to parse CoAP messages and extract useful information. In order to force CoAP requests and responses to be routed through a NAP, CoAP endpoints are configured to use those proxies. Furthermore, the NAPs and the controller are extended with an information *tagging and auditing* system discussed in detail in the following section.

CoAP resources are owned by real world entities, i.e., the resource *owners*. Owners are responsible for selecting which resources should be monitored, as well as for configuring the appropriate entities for this purpose. Owners interact with the system entities through a network endpoint connected (directly or indirectly) to NAP (e.g., a PC, a mobile device, etc). Therefore, whenever we mention in the following that an owner interacts with a particular entity, we always mean through his network endpoint.

4.3 High-Level Operation

From a high-level perspective, our system operates as follows. Owners decide which CoAP resources are sensitive and *register* a *tag* prefix for each CoAP host identifier (see Sect. 2.2) they own. Then, they *configure* the appropriate NAPs with the corresponding tags. Whenever some data associated with a protected resource *flows* through a NAP, the NAP tags it accordingly. Furthermore, SDN switches located at the edges of the network report to the SDN controller the tagged flows they observe. These reports are recorded by an auditing service. Finally, before a tagged flow is delivered to an end user, its tag is removed. Therefore, all operations are transparent to end users. This process is illustrated in Fig. 4.

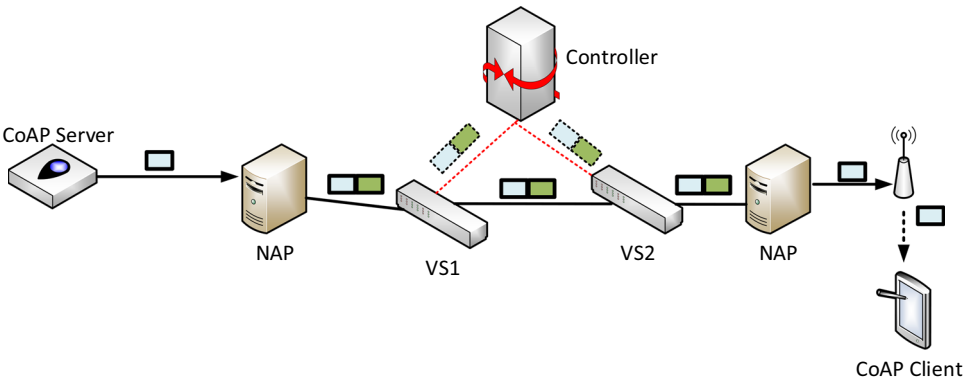

Fig. 4 High-level overview of the proposed system

4.4 Security Considerations

For our solution, we are making the following security-related assumptions. NAPs are considered trusted and they always behave according to the specified procedures. NAPs are owned, controlled and protected by the network operator. They are physically isolated from end users and attackers cannot connect a random NAP to a network. Moreover, we assume an authentication process between NAPs and the endpoints connected to them (e.g., they are configured with a pre-shared secret key). This process, which is out of the scope of this paper, ensures that attackers: (i) cannot connect to an arbitrary NAP and (ii) cannot modify the NAP configuration (related to our system) of another user. Finally, we assume that the communication channel between a CoAP endpoint and a NAP, as well as the NP internal network, are secured (e.g., using "Distributed security for multi-agent systems" [24]).

At a higher layer, our design assumes a secure mechanism that is used by a tag registration service for verifying CoAP host identifier ownership by a particular owner. Additionally, each owner is identified by an NP-specific identifier which can be verified by the tag registration service and the NAPs. We refer to this identifier as the *owner identifier*. The owner identifier can have any form, it can even be a network related identifier.

Given the above security assumptions, our security threat model considers attackers that try to "pollute" the tagging and auditing system with invalid entries. Hence, these attackers are legitimate resource owners that can connect to a NAP and can configure it accordingly.

5 System Design

5.1 Tag Registration

The purpose of the tag registration process is to protect our system from the attacks mentioned in Sect. 4.4.

Prior any operation, each owner must receive a tag prefix (for each CoAP host identifier they wish to protect) with a tag registration service. This service is operated by the NP (network provider). The owner communicates with that service securely and verifies his identity and the host identifier ownership (using–as discussed–a mechanism which is out of the scope of this paper). The output of the registration process is a unique tag prefix and a cryptographic access *token* that binds the tag prefix to: (i) a particular owner identifier and (ii) to a particular CoAP host identifier. The token is digitally signed by the registration service with a signature that can be verified by all NAPs. This token must be kept secret by owners. For additional security, an access token may have an expiration time so that owners refresh it periodically.

5.2 NAP Configuration

Each owner is responsible for configuring the appropriate NAP(s). NAPs are accessed using a remote procedure call (RPC) backend over HTTPS. Owners configure NAPs by modifying a *tags table* that contains tuples of the form $[URI, tag, policy]$, where URI is the CoAP URI of the protected resource, tag is the tag that should be used for tagging it, and $policy$ is a URI matching policy. Currently, our system considers two policies: *prefix match*, and *exact match*. As the name implies, with the first policy, a resource is protected if its URI is prefixed with the URI included in the tags table, whereas with the second policy, a resource is protected if its URI is included in the tags table. If there are two entries in the tags table with the same URI but with different policy, then the exact match policy has higher priority.

NAP configuration is a two-step process: Firstly, the owner connects to the NAP, and then he configures the tags table. In the first step, the NAP should accept a connection only if the owner is authorized to connect to that NAP. This is achieved using NP-specific mechanisms that are out of the scope of this paper. When this step is completed, the NAP is

assumed to have learned the owner identifier. In the second step, the NAP must verify that the owner has the right to perform the requested action to the tags table (add a new line, or delete/modify an existing one). This is achieved using the following procedure:

1. The owner requests to perform a specific action. His request includes the CoAP URI, the access token, and, if required by the action a tag and a policy.
2. The NAP verifies the digital signature of the token, and if the token includes an expiration time, it verifies that the token is still valid.
3. If the requested action includes a tag (e.g., create a new entry in the tags table), the NAP verifies that its prefix is included in the provided access token.
4. The NAP verifies that the token is bound to the owner identifier, as well as to the CoAP host identifier of the URI specified in the action request.

At the end of this procedure, and if all checks are successful, the NAP performs the requested action.

5.3 Message Flows

Since CoAP responses do not include the requested CoAP URI, a NAP should keep track of the CoAP requests. In our system, we assume that all CoAP requests include a token.[1] Whenever a CoAP request that concerns a protected resource is forwarded by a NAP to a CoAP server, the NAP updates a *pending responses table* that contain tuples of the form $[token, tag, type]$, where $token$ is the token included in the CoAP request, tag is the tag that corresponds to the requested CoAP URI, as retrieved from the tags table, and $type$ is the type of message flow. Types affect when an entry is removed from the pending responses table and are discussed in more detail later on in this section. Hence, when a NAP forwards a CoAP request from the network to a CoAP server, it performs the following procedure:

1. It retrieves the CoAP URI and the token from the request message
2. It checks if the CoAP URI is included in the tags table (which means that this is a protected URI which should be monitored)
3. If the CoAP URI is protected, it retrieves the tag from the tags table, and creates a new entry in the pending responses table

Whenever a CoAP response is forwarded from a NAP to the network, the NAP retrieves the token from the response, examines the pending responses table, and if it includes the retrieved token it tags the transmitted packet and forwards it to the network. When a tagged packet reaches the destination NAP, the NAP removes that tag.

We distinguish five types of messages flows: (i) confirmable requests followed by piggybacked responses, (ii) non-confirmable group communication requests followed by non-confirmable responses, (iii) non-confirmable requests or confirmable requests followed by separated non-confirmable response, (iv) non-confirmable requests or confirmable requests followed by separated confirmable response, and (v) CoAP observe request followed by resource updates. For the first three types, a NAP removes immediately the corresponding entry in the pending responses table. For type IV, the NAP removes the entry when it received the acknowledgement from the CoAP client. Finally, for type V, the NAP removes the entry when the client sends a deregister message.

5.4 Tracking and Auditing

Each SDN switch connected to a NAP is configured with OpenFlow rules that instruct it to send a copy of the tagged packets they forward to the SDN controller (we provide details how this is achieved in Sect. 6). The SDN controllers store this information in an *audit table* the rows of which are composed of tuples that contain the tag, the identifier of the incoming switch, the identifier(s) of the outgoing switch(es), a time stamp as well as a hash of the packet. Furthermore, each NAP that forwards a (tagged) response to a CoAP client informs the auditing service about the destination IP address(es). This information is also recorded in the audit table.

In case of a security incident, an owner can consult the auditing service and determine the addresses of the CoAP clients that received a packet. In order to achieve that he must query the audit table using the access token and the tag of the item in question (since the audit table does not contain the CoAP URI of the tracked packets).

6 Implementation and Evaluation

We have implemented the proposed solution using the Open vSwitch software SDN switch [25] and the POX SDN controller [26]. As NAPs, we used standard Linux-based PCs and we implement our CoAP endpoints and CoAP proxies using the libcoap library.[2] Since packet forwarding takes place using a Bloom filter stored in the IPv6 address fields, the Ethernet address fields are not used at all. For this reason, we are storing our tags in the Ethernet destination address field, which gives us space for 2^{48} tags.

[1] This assumption can be easily fulfilled since all CoAP requests are sent through our proxy, which can add a token if there is not one.

[2] https://github.com/obgm/libcoap.

Packet tagging at NAPs takes place as follows. The NAP replaces the packets Ethernet source address with "00:00:00:00:00:01," to indicate that this is a tagged packet, and the Ethernet destination address with the tag. SDN switches attached to a NAP are configured with an OpenFlow rule that instructs them to forward flow to an SDN controller as an Openflow *PacketIn* [27] message if: (i) The flow originates from or has a destination a NAP,[3] and (ii) its Ethernet source address is 00:00:00:00:00:01. The NAP that forwards the tagged response to the CoAP client replaces the Ethernet source address with the NAP's address and the Ethernet destination address with the client's address (or the appropriate Ethernet destination address, in case the client is not directly connected to the NAP).

6.1 Overhead Evaluation

The overhead imposed by our solution is mainly due to the pending responses tables and the copies of the packets forwarded from the SDN switches to the controllers. In order to measure this overhead, we consider the workload of the temperature measurement sensors deployed in the testbed of SmartSantander, a large-scale smart city deployment located in Santander, northern Spain. The workload, as reported in [28], is composed of 70 temperature sensors, generating a temperature measurement approximately every 5 min. We consider the following extreme (and highly unlikely) case: All sensors are connected to the network through the same NAP, all sensors produce a new measurement simultaneously, for each sensor measurement a different tag is used, and CoAP clients are using the CoaP observe extension and have register their interest in all measurements. We assume that each client-side NAP can accommodate up to 10 CoAP clients (hence, if there are 10 clients in our experiments they are connected through 10 NAPs). Furthermore, we consider the following cases:

- Optimization 0: No optimization is applied
- Optimization 1: Client-side NAPs aggregate the requests of its CoAP clients (therefore, there is a single registration per NAP)
- Optimization 2: The optimizations of [22] are used (therefore, measurement updates are transmitted using a single packet)

Figure 5 shows the size in bytes of the pending responses table as a function of the number of CoAP clients. For this experiment, we have set the size of tag to 6 bytes, the size of a token to 8 bytes (this is the maximum value as specified in [3]), and the size of a policy to 1 byte. The size of the table when Optimization 2 is applied is not shown since it is constant and equal to 15 bytes (i.e., when this optimization is applied the pending responses table includes a single entry).

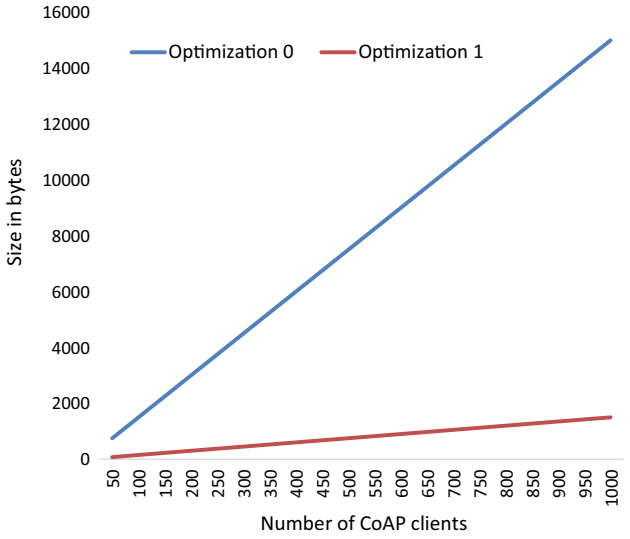

Fig. 5 Size of the pending responses table as function of the number of CoAP clients

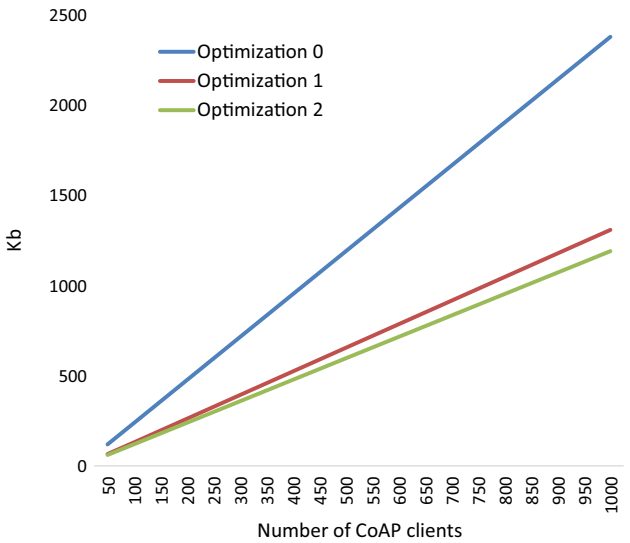

Fig. 6 Size of total messages transmitted to the controller

Figure 6 shows the size of messages sent to the controller with every update. For this experiment, we have set the size of a packet to 128 bytes (based on the size of the packet headers and the recommended maximum CoAP message size as specified in [3]). It should be noted that these messages concern bursty traffic; hence, although the average throughput of the link to the controller should be low, its peak will be almost equal to the total size of messages per update (as depicted in Fig. 6).

[3] This is determined based on the incoming/outgoing network interface.

7 Conclusions

This paper presented an SDN-based solution for information flow tacking and auditing for the IoT. The presented solution is lightweight since it adds minimum storage overhead (in the order of Kbytes per NAP), as well as insignificant network overhead (in our worst case, unrealistic scenario the network overhead was measure to 2 Mbytes per 5 min). Furthermore, the proposed solution is realistic since it requires no modifications to CoAP endpoints and operates using standard OpenFlow operations (i.e., no modifications are required to the SDN switches).

In this work, we selected CoAP as our reference IoT protocol. However, this work can be extended to include other IoT protocols such as Message Queuing Telemetry Transport (MQTT). Of course, our solution can be considered for similar protocols used in other contexts (e.g., HTTP), as well as for emerging information-oriented communication paradigms (e.g., the Information-Centric Networking paradigm). Our work considers only CoAP over UDP; however, other transport protocols can be considered for CoAP (e.g., TCP or even SMS); in order to cope with this transport protocols, our solution has to be modified accordingly. Furthermore, many products consider CoAP over DTLS. This approach prevents the use of our solution since it hides the CoAP payload. For this reason, in order for our solution to be used with DTLS, encryption interception mechanisms should be used (as used for example in [29]).

Acknowledgements This Project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under Grant No. (D-311-611-1440). The author, therefore, gratefully acknowledge the DSR for technical and financial support.

References

1. Khan, M.A.; Salah, K.: Iot security: review, blockchain solutions, and open challenges. Future Gener. Comput. Syst. **82**, 395–411 (2018)
2. Xia, W.; Wen, Y.; Foh, C.H.; Niyato, D.; Xie, H.: A survey on software-defined networking. IEEE Commun. Surv. Tutor. **17**(1), 27–51 (2015). https://doi.org/10.1109/COMST.2014.2330903
3. Shelby, Z.; Hartke, K.; Bormann, C.: The constrained application protocol (CoAP). RFC 7252, IETF (2014)
4. Lara, A.; Kolasani, A.; Ramamurthy, B.: Network innovation using openflow: a survey. IEEE Commun. Surv. Tutor. **16**(1), 493–512 (2014). https://doi.org/10.1109/SURV.2013.081313.00105
5. Jarschel, M.; Wamser, F.; Hohn, T.; Zinner, T.; Tran-Gia, P.: Sdn-based application-aware networking on the example of youtube video streaming. In: 2013 Second European Workshop on Software Defined Networks (EWSDN). IEEE Computer Society, Los Alamitos, CA, USA (2013). https://doi.org/10.1109/EWSDN.2013.21
6. Mekky, H.; Hao, F.; Mukherjee, S.; Zhang, Z.L.; Lakshman, T.: Application-aware data plane processing in sdn. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN '14, pp. 13–18. ACM, New York (2014). https://doi.org/10.1145/2620728.2620735.
7. Fotiou, N.; Siris, V.A.; Xylomenos, G.; Polyzos, G.C.; Katsaros, K.V.; Petropoulos, G.: Edge-icn and its application to the internet of things. In: 2017 IFIP Networking Conference (IFIP Networking) and Workshops, pp. 1–6 (2017). https://doi.org/10.23919/IFIPNetworking.2017.8264880
8. Rahman, A.; Dijk, E.: Group communication for the constrained application protocol (CoAP). RFC 7390, IETF (2014)
9. Hartke, K.: Observing resources in the constrained application protocol (CoAP). RFC 7641, IETF (2015)
10. Portokalidis, G.; Slowinska, A.; Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. SIGOPS Oper. Syst. Rev. **40**(4), 15–27 (2006). https://doi.org/10.1145/1218063.1217938
11. Qin, F.; Wang, C.; Li, Z.; Kim, H.; Zhou, Y.; Wu, Y.: Lift: A low-overhead practical information flow tracking system for detecting security attacks. In: 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), pp. 135–148 (2006). https://doi.org/10.1109/MICRO.2006.29
12. Zhu, D.Y.; Jung, J.; Song, D.; Kohno, T.; Wetherall, D.: Tainteraser: protecting sensitive data leaks using application-level taint tracking. SIGOPS Oper. Syst. Rev. **45**(1), 142–154 (2011). https://doi.org/10.1145/1945023.1945039
13. Attariyan, M.; Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pp. 237–250. USENIX Association, Berkeley, CA, USA (2010). http://dl.acm.org/citation.cfm?id=1924943.1924960. Accessed 22 July 2019
14. Kemerlis, V.P.; Portokalidis, G.; Jee, K.; Keromytis, A.D.: Libdft: practical dynamic data flow tracking for commodity systems. SIGPLAN Not. **47**(7), 121–132 (2012). https://doi.org/10.1145/2365864.2151042
15. Enck, W.; Gilbert, P.; Han, S.; Tendulkar, V.; Chun, B.G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. **32**(2), 5:1–5:29 (2014). https://doi.org/10.1145/2619091
16. Gordon, M.I.; Kim, D.; Perkins, J.; Gilham, L.; Nguyen, N.; Rinard, M.: Information-flow analysis of android applications in DroidSafe. In: Proceedings of Network and Distributed System Security Symposium (NDSS) (2015)
17. Bawany, N.Z.; Shamsi, J.A.; Salah, K.: Ddos attack detection and mitigation using sdn: methods, practices, and solutions. Arab. J. Sci. Eng. **42**(2), 425–441 (2017)
18. Salah, K.; Alcaraz Calero, J.M.; Zeadally, S.; Al-Mulla, S.; Alzaabi, M.: Using cloud computing to implement a security overlay network. IEEE Secur. Priv. **11**(1), 44–53 (2013)
19. Al-Haidari, F.; Sqalli, M.; Salah, K.: Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, vol. 2, pp. 256–261 (2013)
20. Calyam, P.; Rajagopalan, S.; Seetharam, S.; Selvadhurai, A.; Salah, K.; Ramnath, R.: Vdc-analyst: design and verification of virtual desktop cloud resource allocations. Comput. Netw. **68**, 110–122 (2014). Communications and Networking in the Cloud
21. Pappas, V.; Kemerlis, V.P.; Zavou, A.; Polychronakis, M.; Keromytis, A.D.: Cloudfence: Data Flow Tracking as a Cloud Service, pp. 411–431. Research in Attacks, Intrusions, and DefensesSpringer, Berlin (2013)
22. Fotiou, N.; Mendrinos, D.; Polyzos, G.C.: Edge-assisted traffic engineering and applications in the iot. In: Proceedings of the 2018 Workshop on Mobile Edge Communications, MECOMM'18, pp. 37–42. ACM, New York (2018). https://doi.org/10.1145/3229556.3229561.
23. Reed, M.J.; Al-Naday, M.; Thomos, N.; Trossen, D.; Petropoulos, G.; Spirou, S.: Stateless multicast switching in software defined

networks. In: 2016 IEEE International Conference on Communications (ICC), pp. 1–7 (2016). https://doi.org/10.1109/ICC.2016.7511036
24. Rashvand, H.F.; Salah, K.; Calero, J.M.A.; Harn, L.: Distributed security for multi-agent systems—review and applications. IET Inf. Secur. **4**(4), 188–201 (2010). https://doi.org/10.1049/iet-ifs.2010.0041
25. Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P.; Amidon, K.; Casado, M.: The design and implementation of open vswitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pp. 117–130. USENIX Association, Oakland, CA (2015)
26. Gude, N.; Koponen, T.; Pettit, J.; Pfaff, B.; Casado, M.; McKeown, N.; Shenker, S.: NOX: towards an operating system for networks. SIGCOMM Comput. Commun. Rev. **38**(3), 105–110 (2008)
27. OpenFlow Switch Specification v1.2.0 (2011). https://www.opennetworking.org/. Accessed 18 July 2019
28. Siris, V.A.; Fotiou, N.; Mertzianis, A.; Polyzos, G.C.: Smart application-aware iot data collection. J. Reliab. Intell. Environ. **5**(1), 17–28 (2019)
29. Fotiou, N.; Xylomenos, G.; Polyzos, G.C.: Securing information-centric networking without negating middleboxes. In: 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5 (2016)