A Network Service for Preventing Data Leakage from IoT Cloud-assisted Equipment

Vitor A. Cunha*, Eduardo da Silva*†, Marcio B. de Carvalho‡, Daniel Corujo*, Joao P. Barraca*, Diogo Gomes*, Alberto E. Schaeffer-Filho‡, Carlos R. P. dos Santos§, Lisandro Z. Granville‡, Rui L. Aguiar*
*Instituto de Telecomunicações, Portugal
†Department of Informatics – Catarinense Federal Institute – Araquari, Brazil
‡Institute of Informatics – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
§Department of Applied Computing – Federal University of Santa Maria – Santa Maria, Brazil
*{vitorcunha,dcorujo,jpbarraca,dgomes,ruilaa}@av.it.pt, †eduardo.silva@ifc.edu.br,
‡{mbcarvalho,alberto,granville}@inf.ufrgs.br, §csantos@inf.ufsm.br

***Abstract*—The fact that most IoT solutions are provided by third parties, along with the pervasiveness of the collected data, raises privacy and security concerns. There is a need to verify which data is being sent to the third party, as well as preventing those channels from becoming an exploitation avenue. We propose to use existing API definition languages to create contracts which define the data that can be transmitted, their format and constraints. To verify the compliance with these contracts, we propose a Network Service architecture which validates REST-like API requests/responses against a Swagger schema. We deal with encrypted traffic using an Service Function Chaining (SFC)-enabled Man-in-the-Middle (MITM), allowing verifications in "real-time." We devised a Proof of Concept and showed that we were able to detect (and stop) contract violations.**

I. Introduction

The pervasiveness of the data gathered by solutions of the Internet of Things (IoT) allows tackling today's challenges holistically, as seen in Smart Cities, Smart Crops, or richer augmented reality interactions [1]. The IoT systems are composed of appliance-like physical devices *(things)* and network connectivity. By collecting, exchanging, and acting upon data the *things* can deliver smarter, more personalized services and applications [2]. Over the last years, these IoT systems started being built using cloud-based infrastructures to cope with the increasing number of services offered [3], as the cloud significantly improved a multitude of IoT-centric operations, such as service management, data storage, or analysis of data [4]. The IoT gateway equipment became commonplace, bridging different communication technologies to the *things*, aggregating data, and then sending it to the cloud.

In such scenarios, the IoT solutions are usually controlled by a third-party, which installs its equipment in the premises of the costumer, and then uses its cloud to power the IoT application. Encryption is typically employed to safeguard privacy when sending data to the cloud, but this also hinders the ability to audit, because the end-user usually does not hold the keys. Consequently, there is the threat that a solution provider (who may have been compromised) might abuse the regular communications between the IoT equipment and the cloud to exfiltrate more data than allowed or even subvert this channel to do Remote Code Execution (RCE) in its customer-premises equipment, facilitating the exploitation of other machines in customer's local private network(s).

To address these issues, without fundamentally rebuilding the existing IoT systems to be privacy-preserving by design, with auditable Open-Source code, and enhanced anti-exploitation mechanisms, we propose a network-level safeguarding approach. The core component of this proposal is the Contract Validator Network Function (NF), which can inspect the messages between the IoT cloud and the gateway(s) in the user's premises against a previously agreed upon contract, negotiated between the end-user and the IoT provider. To model the contract, and describe the data validation mechanisms, we employ the same specification languages used for Application Programming Interface (API) documentation and compliance, such as Swagger or RAML. Thus, professionals can easily audit the reach of the terms, while our function can automatically validate the API's requests/responses according to the contract, detecting and stopping violations, such as attempts to leak information or facilitate RCE.

The crux to any network-level approach is how to reliably intercept the traffic and ensure the validation NF can verify it. We propose an Network Service (NS) which packages our Contract Validator function together with an SFC-enabled Man-in-the-Middle (MITM) [5], that acts as a generic SSL/TLS offloader, allowing our solution to verify HTTPS encrypted calls, as well as chaining any additional network functions. The NS design allows our architecture both to integrate with the costumer's infrastructure or to offload its deployment to a Multi-Access Edge Computing (MEC)-like environment. The latter may be more suitable for power constrained premises (*e.g.*, remote location that only has battery power), as the power consumption should not change significantly with the introduction of the NS. The MEC-like environment also allows tackling the traffic interception issue regardless of the topology on the customer's premises, in a Security-aaS model. Additionally, it provides cloud computing capabilities (to scale instantly and on-demand) and built-in

978-1-7281-2999-0/19/$31.00 ©2019 IEEE

traffic offloading functions. Because MEC is right at the edge of the consumer's network, it allows for good perimeter defense (if operated by a trusted fourth-party). Nevertheless, as long as the premises are not power constrained, the same feature-set is still possible within the customer infrastructure, just at a higher effort.

The paper is structured as follows. First, we will briefly introduce the current alternatives to tackle this problem and present background information about our approach (Section II). Then we will present our solution architecture (Section III). The Proof of Concept and evaluation results are described in Section IV. Finally, we present the conclusions (Section V).

II. Related Work and Background

In this section, we will start by presenting the different approaches which were already proposed to address the need to audit data in existing IoT systems. Then, we will present new cryptographic technologies that allow redesigning IoT systems to be privacy preserving. Finally, we present some background which relates to the network-level approach.

A. Related Work

The TLS-Rotate and Release (TLS-RaR) [6] system provides a way to audit IoT traffic, without having to modify the TLS protocol and disrupt the IoT solution, but at the cost of traffic inspection not being done in real-time. It requires cooperation from the IoT solution provider, but that cooperation may be selective. For instance, it may use additional encryption (to hide data/activities) when it suits its interests.

In turn, the Multi-Context TLS (mcTLS) [7] protocol allows real-time traffic inspection (and modification). Previously authorized middleboxes will be given an additional key which will grant them either read-only or read/write permissions over the flow. Despite having as drawback a compatibility break with the current TLS endpoints, mcTLS derivate protocols like Transport Layer MSP (TLMSP) are already being drafted by normative bodies such as European Telecommunications Standards Institute (ETSI). Much like TLS-RaR, these protocols also require cooperation from the IoT solution provider.

In order to attempt to bypass the need for cooperation with the IoT solution provider, more direct TLS MITM solutions exist, such as BlindBox [8], SGX-Box [9], or mitmproxy [10]. These solutions either exploit the faulty verification of trusted roots or require the insertion of a self-signed certificate as a trusted root. This raises some concerns, as failure to comply with proper Certification Authority (CA) practices may pave the way for malicious actors to gain an even larger foothold over the data transacted within the network. A practical concern is that the NFs performed within these middleboxes need specific development for that middlebox. We have tackled this issue in [5] using an Service Function Chaining (SFC)-enabled MITM which allowed to process HTTPS traffic in functions that already worked as plain HTTP proxies (decoupling the SSL/TLS layer from the actual HTTP contents).

Newer cryptographic technologies, such as blockchain and Smart Contracts as in the Ethereum framework [11], allow building entirely new IoT platforms that may already have different built-in checks over the transacted data.

B. Background

Network Function Virtualization (NFV) brings cloud computing into the Telecom world, allowing for greater flexibility, lower time-to-market, and ways to reduce the Capital Expenditure (CAPEX) and Operating Expense (OPEX) of operators. The ETSI specifies an NFV framework [12] which normalizes the requirements, interfaces, and description of virtualized NFs. More than just virtualization of functions, ETSI NFV paves the way to the Management and Orchestration (MANO) of complex automatic, self-provisioned, and on-demand NSs (as normalized in ETSI NFV-MANO [13]).

The ETSI MEC architecture [14] allows to run applications closer to the end-user (the Edge), improving performance with latency sensitive applications, providing consistent user experience (*e.g.*, late transmuxing), reducing the back-haul utilization (*e.g.*, content caches), and enhancing network security by stopping attacks when they are ingressing the network. MEC is an integral part of 5G technologies. Earlier works already shown the use of NFV alongside MEC [15], leveraging the Traffic Offloading Function (TOF) to run network services on the Edge. Lastly, the composition of an SFC by concatenating NFs in a given order has already been standardized by IETF [16].

III. The Approach

We propose an NS to validate the IoT gateway communications (with the Cloud) against a previously agreed specification. We start by presenting our IoT message validation architecture, also describing the means to define the contract used for message validation. Then, we propose an approach to validate the contract, which is our Contract Validator NF, that completes the NS required functionality.

A. Message Validation Architecture

Our main contribution is the NS as shown in Fig. 1, which depicts an architecture allowing to open SSL/TLS ciphered communications (through an MITM NF, since we may not

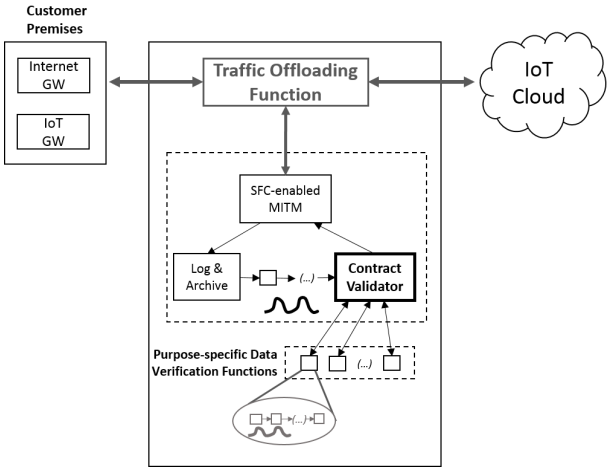


Fig. 1: Proposed IoT messages validation architecture

rely on IoT provider cooperation in our solution) and then processes them in plain-text by an SFC of NFs. Since most gateway to cloud communication is usually done through a REST API, it is at stake handling HTTP(S) in the SFC-enabled MITM (which was already explored in [5]). Each NF will only handle the part of the HTTP stack that it needs to process, minimizing data exposure and overheads. As an NS, the architecture may be deployed both on user's premises (enterprise users) or on a delegated infrastructure (MEC-like or cloud). Because some locations may have electric power constraints, such as being battery powered, our proposal emphasizes the option of offloading the verification functions to a fourth-party Edge/Cloud (independent of the contracted third-party IoT solution provider) as a strategy to avoid increasing the power consumption in the premises of the IoT system.

The rise of Test Driven Development (TDD) methodologies, and its applications in DevOps, already drove the need to create specification languages defining the desired functionality, as well as tools to verify the compliance of code/functions against a given specification. We propose to take advantage of the same API specification languages that are already available to test the system during its development. This will minimize development overheads, while still providing means to reliably verify the implementation against the well-behaving interface specification (called schema). If the schema only allows secure and private communications, then that becomes a verifiable contract that prevents abuse from the IoT vendor. Therefore, a trusted NS can enforce the contract by verifying the API interactions against the schema. In turn, by being built as an NS, the architecture allows offloading the contract verification to a trusted party, removing additional hardware costs or large impacts over existing IoT solutions.

In order to define the verifiable contracts, we propose the use of Swagger. This enables specifying the allowed API endpoints, paths, methods, expected headers, and key-values. Swagger also permits some data verifications, such as numeric boundaries checks, or value-based format verifications.

Tools such as Swagger-Proxy allow validating *responses* from a running REST service which has its API schema published. However, only verifying the response messages would be unsuitable for our purpose, as the data being sent from the IoT gateway to the cloud can be exfiltrated without any check. Thus, in order to enforce the verification of HTTP Requests, we developed a new NF. Besides, particular data-types that may require additional validation that would fall outside of the capabilities of a schema validator. So, we proposed an extra layer of NFs that communicate directly with the *Contract Validator* and perform data specific validations. We chose to make clear that each of these NFs could themselves be an SFC of more specialized units.

In the next subsections (III-B and III-C) we will further elaborate the way the IoT contract can be defined using this schema, as well as define how the *Contract Validator* must function in order to prevent API abuse.

B. Defining the Contract

We propose the use of Swagger to define the interface contracts between the IoT customer and the IoT solution provider. These are already *de facto* standards to disseminate the API documentation/specification across multiple parties. The schema can define all paths, methods, parameters, headers, accepted content formats, most authentication requirements, the host that must be contacted, API versioning and allowed transport protocols. It also provides the most usual verifications over data-types, such as boundaries check (in integers and floats), enumerated items, and format checks (for strings).

Definition 1 presents an excerpt from the Swagger schema created to evaluate our proposal. In this example, it is described the POST method **sendTemperature** of a given path **/temperature**. By invoking this method, the IoT sensors can feed the cloud with the current temperature data.

Note that on line 17 of Def. 1, there is a reference to **#/definitions/TemperatureData** to define the parameters' schema of the path. Its definition is presented on Def. 2, in which the temperature data must provide two fields, **id** and **value**. The former must be a string in UUID format, while the latter must be a float number, ranging from **-99.9** to **99.9**.

```
1   swagger: "2.0"
2   paths:
3   /temperature:
4   post:
5   tags:
6   - "temperature"
7   operationId: "sendTemperature"
8   consumes:
9   - "application/json"
10  produces:
11  - "application/json"
12  parameters:
13  - in: "body"
14  name: "body"
15  required: true
16  schema:
17  $ref: "#/definitions/TemperatureData"
18  responses:
19  200:
20  description: "OK"
21  (...)
```

Def. 1: Excerpt from Swagger schema

```
1   definitions:
2   TemperatureData:
3   type: object
4   required:
5   - "id"
6   - "value"
7   properties:
8   id:
9   type: string
10  format: uuid
11  value:
12  type: number
13  format: float
14  minimum: -99.9
15  maximum: 99.9
```

Def. 2: TemperatureData definition

C. Verifying the Contract

We have designed a Contract Validator NF (Fig. 2) which, before doing any standard schema validation over the messages arriving from the chain, performs a whitelist check for every key, parameter name, and header of the message. This procedure allows tackling abuses through the use of extra data (or meta-data) which may not break the specification, while it still allows leveraging the standard validation tools to perform the checks that fall within the standard specification validation.

While the combination of the whitelisting with the standard schema checks already delivers an effective mechanism to verify the compliance of the contract, there is still one fundamental limitation to beware. The validation process verifies if the data is in a valid format, according to the specification. However, that does not mean the data is correct or untampered. For instance, an adversary may use steganography to establish a covert channel, encoding rogue data over a valid format [17]. Additionally, we may have a stream of data (audio, video, or other binary data) which, despite being valid in its formatting, is still leaking private information.

Thus, our design allows using external purpose-specific functions to perform further validations after the standard schema checks. Sample functions could be a Kalman Filter to protect user privacy [18], a method to detect LSB steganography on images [19], the IIoT-SIDefender to detect and defend against sensitive information leakage [20], or others. The MEC-like offloading allows chaining these more powerful verification functions on-demand, as resources can be pooled like in cloud computing. We can even add extra functionality without disrupting the system. For instance, we could live-test machine learning classification [21] functions.

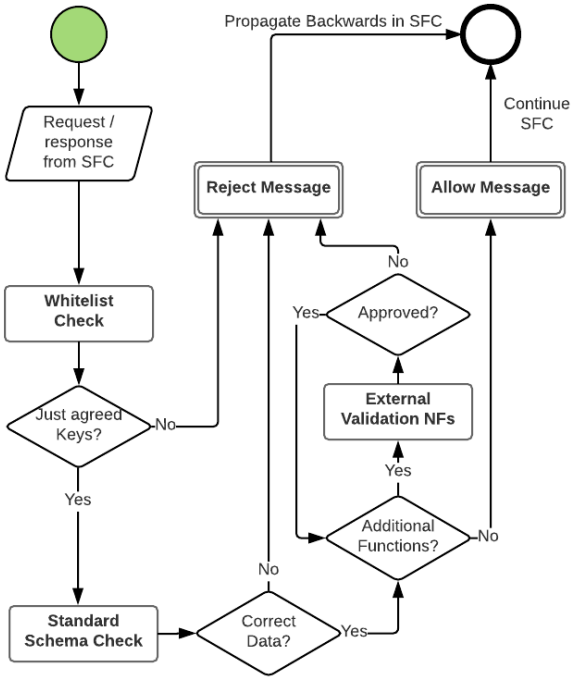


Fig. 2: Proposed Contract Validator Function

IV. Evaluation

In this section, we detail the design and implementation of our PoC, along with its metrics, and finally the results.

A. Proof of Concept

1) Scenario: Our Proof of Concept (PoC) considers a smart-home scenario where a cloud-assisted IoT application adjusts a window's blinds position according to the data gathered by temperature and luminosity sensors. The objective is to optimize sunlight, *e.g.*, heating the room in the winter.

The smaller sensors (and actuators) use heterogeneous access technologies, relying on the IoT gateway to communicate with the cloud. The gateway gathers the data from sensors and sends it to the cloud, using the defined APIs. Then, the IoT application processes the received data and makes decisions (*e.g.*, blinds target position). Also, the IoT gateway is aware of the application and polls the cloud API – with a variable rate – to adjust the position of the blind to a given target.

2) Evaluated Exploit: We modeled our qualitative evaluation after the exploitation of the *EthernalBlue* vulnerability[1], against a victim system within the home's private network. Being a more complex attack, that requires pivoting to a private network and exploiting an otherwise hidden target, it highlights throughout its different stages different kinds of IoT API abuses, which the Contract Validator Function should stop. The scenario was evaluated as depicted in Fig. 3. Following the API's Def. 1 and 2, Fig. 3a shows the expected interaction. Fig. 3b to Fig. 3d illustrate the three stages of a pivoting attack that abuses the first valid interaction.

The IoT gateway has "undocumented features" which allows the provider to execute a command sent in the key *cmd*, as well as overriding the next value to be sent to the Cloud with exfiltration data stored in the sample */tmp/exfiltrate.file*. For brevity, the base64 encoded data has been replaced with shortened dummy values. In the first stage (Fig. 3b), the provider leverages an extra key *(invalid as per contract)* in the response message to execute a *nmap*[2] command which finds all machines in the private network which are vulnerable to *EthernalBlue*, and stores the output in the exfiltration file. In the second stage (Fig. 3c), the temperature reading of the sensor is replaced with a base64 encoded version of the exfiltration file, which contains all vulnerable machines within the private network. Lastly, in the third stage (Fig. 3d), the provider uses the information gathered in the second stage to send in-line with the command a base64 encoded malicious payload that will exploit one of the targets. This is made easy using the Metasploit framework[3] and adapting publicly available PoCs[4]. In particular, a reverse meterpreter shell payload will allow direct access to the target machine, having the highest system privileges available to execute commands in its victim (due to the nature of the *EthernalBlue* vulnerability).

[1]https://technet.microsoft.com/library/security/MS17-010
[2]https://nmap.org
[3]https://www.metasploit.com/
[4]https://github.com/worawit/MS17-010

3) Implementation: We have performed our PoC using containers (LXD) in a larger VM (8 vCPUs with 8 GiB RAM). We had four containers: the gateway, the simulated cloud, the SFC-enabled MITM [5], and the Contract Validator.

The Contract Validator function was implemented in Python 3, being the HTTP socket handling done with the built-in HTTP module (acting as a proxy), the white-listing process done with custom code, and the standard validation performed with a slightly modified FLEX library [22].

For evaluation, the IoT gateway simulated the sensor events, data, and subsequent calls to the cloud. The API used by the gateway to communicate with the cloud is built atop HTTP(S) in a REST-like fashion. The IoT providers' cloud was built using a REST server auto-generated from the Swagger descriptor, in Python-Flask code. This server reacts with crafted responses for each method of the API. By the end, SSL/TLS was added to this server using Nginx as a reverse proxy.

Although our PoC relies on HTTPS as an underlying protocol for communication, the architecture and concepts discussed in Section III could be used to build similar systems for the IoT APIs that rely on proprietary protocols for communication. However, departing from JSON and REST-like interfaces causes significant shortcomings in the definition languages available to define the contracts, and will require extra work to perform the automatic verification of the contract.

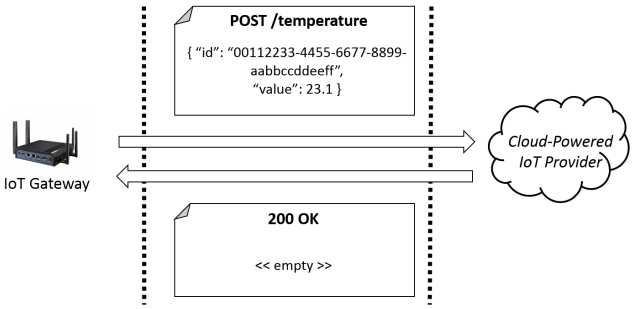
B. Results

We focused the evaluation on our main contribution, the Contract Validator NF. The other architectural modules were either already evaluated (*e.g.*, traffic offloading [15]) or too specific to the particular deployment (*e.g.*, extra validation functions). We considered two metrics to evaluate the Contract Validator: effectiveness and processing time. As HTTPS is widely adopted to secure these communications, we have also evaluated the impacts caused by using the Contract Validator in conjunction with the SFC-enabled MITM. Therefore, a third metric arose to evaluate this scenario: end-to-end delay.

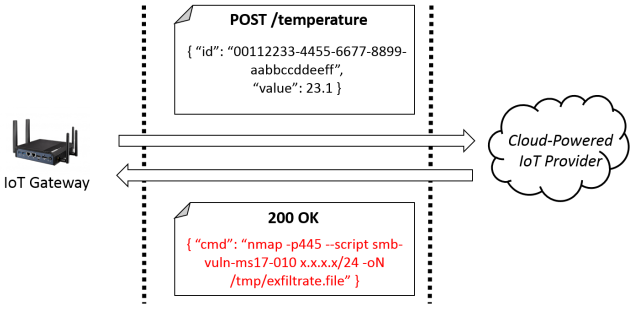
The **effectiveness** is qualitative, to prove that the solution can stop data-leakage or RCE attempts made through the abuse of the API. Thus, we need to verify whether the solution stops the requests containing the exploits detailed in Section IV-A2.

The Contract Validator was effective in stopping the exploitation attempts of the scenario. All requests (and replies) which contained more keys than allowed in the specification were blocked. It was also able to stop the messages which had data in a non-compliant format. Blocked messages events were signaled back by returning the HTTP status code 403.

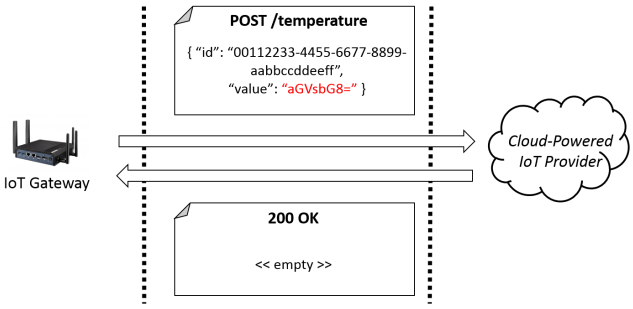
Processing time measures the times each operation took inside the Contract Validator function. Three meta-times were measured: building the whitelist, whitelist checking, and standard schema check. **Build whitelist** is the time to build, from the swagger schema, the proper keys whitelist for the incoming message. **Whitelist Check** is the time to process the request/response and check if all keys complied with the whitelist. Finally, **Standard Schema Check** time represents the time to validate the whole request/response.



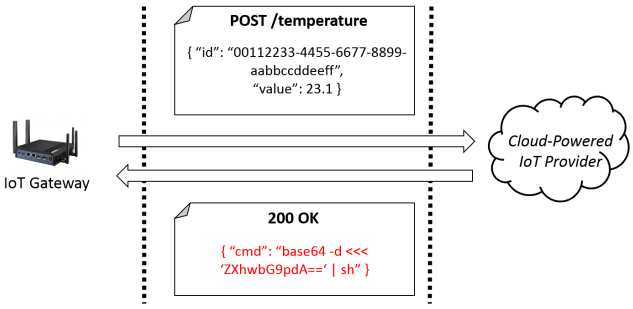
(a) The expected interaction with the provider's Cloud (send the temperature measured by a given sensor)



(b) Attack Stage 1: Provider uses the response of a valid interaction to send a rogue command to the gateway



(c) Attack Stage 2: Provider uses a valid request field to exfiltrate rogue data (*e.g.*, targets list)



(d) Attack Stage 3: Provider uses Remote Code Execution (RCE) to run an in-line encoded exploit against a target

Fig. 3: Qualitative Evaluation Scenario

The results of the processing time are presented in a plot, as shown in Fig. 4. The left portion shows valid requests with different data-types. The right portion shows the measurements taken with requests that were valid, added an extra key, or

Fig. 4: Processing Times inside the proposed Contract Validation NF (*ms*)

Fig. 5: End-to-End Connection Times using HTTPS encrypted flows (*ms*)

had valid keys with invalid data. Such scenarios are denoted as **valid info.**, **invalid key**, and **invalid data**, respectively. The experiments were repeated 100 times each, bars show the average value, interval shows the distance to min./max., and outliers of the 95% confidence interval are not considered.

Note that building the whitelist is proportionally so fast (0.014 ± 0.006 ms) that its graphical representation is virtually invisible in the plot. The results have shown that building the whitelist takes about the same time regardless of data type, or the validity of the message to be checked. The whitelist check takes the same time when all keys are in the whitelist. However, it will be slower when it has to block that transaction (0.27 ms vs. 0.04 ms), due to termination overheads of the inspection for that flow and the generation of the block message. Finally, the standard schema check execution times are very similar regardless of the message validity (1.76 ms for valid data vs. 1.87 ms with non-compliant content).

The last metric, **end-to-end delay** measures the time elapsed after the request is sent until the response is received. The aim is to measure the impact of performing the MITM plus the outbound message validation solution when processing HTTPS flows. We considered three times: direct, sfc-short, and sfc-swag. **Direct** corresponds to the regular end-to-end Round-Trip Time (RTT). **Sfc-short** denotes the RTT when adding the SFC-enabled MITM solution to open encrypted data and re-encrypt before sending to the destination. Finally, **sfc-swag** measures the end-to-end RTT when using our message verification solution in conjunction with the SFC-enabled MITM.

The evaluation results about end-to-end delay are presented in Fig. 5. Like before, the left portion of the plot shows the impact on the end-to-end delay when transmitting different data-types. The right portion shows the measurements considering also requests that were valid, but compared to those that had extra keys or invalid data. Such scenarios are denoted as **valid info.**, **invalid key**, and **invalid data**, respectively. The experiments were repeated 100 times each, bars show the average value, interval shows the distance to min./max., and outliers of the 95% confidence interval are not considered.

The results show that the transmission of different data-types does not present a significant time variance on end-to-end delay within the same scenario. We can note this in the direct (around 9 to 10 ms), sfc-short (around 43 to 45 ms), and sfc-swag (around 49 to 51 ms) scenarios. This observation confirms that the previous findings of the processing time of different data-types also hold true when the Contract Validator is deployed with SFC-enabled MITM.

When introducing the SFC-enabled MITM (sfc-short), we observe that the times rise to above 40 ms. This overhead was expected given the additional cryptographic processing, which includes generating on-the-fly server certificates to impersonate the IoT cloud. This could be avoided if the IoT solutions adopted secure communication protocols more suitable for middlebox inspection, such as the mcTLS. However, as the use of such protocols is not yet widespread amongst existing IoT vendors, we only evaluated the standard TLS in HTTPS. Thus, by using already in place protocols, our proposal evaluates more accurately the reality of adding verification capabilities to the existing TLS based IoT solutions.

Finally, adding our message verification solution (sfc-swag), we see an increase of about 5 ms, which is consistent with the previous results (Fig. 4) as we have double processing time (request + response) and a few connection handling overheads. It is important to note that despite a considerable rise in the end-to-end time comparing direct access to sfc-swagger (from 10 ms to 50 ms), most of the overhead (around to 35 ms) was introduced by the SFC-enabled MITM. This observation highlights the need for a more suitable and standardized HTTPS middlebox traffic inspection protocol.

We then evaluated our solution in scenarios where compromised or malicious internal devices are sending non-compliant data to the cloud of the solution provider. We considered three scenarios (right portion of Fig. 5). Note that when our message verification solution blocks a request (*invalid key* and *invalid data* scenarios), the end-to-end time will be half the regular time, approximately 25 ms. This is an expected consequence of blocking a request message, the fact that the remote cloud is no longer contacted, therefore all those overheads avoided.

The proposal was effective at stopping the abuse of the communications API, as shown by the multi-stage attack (exploiting the *EthernalBlue* vulnerability) performed against

an otherwise inaccessible target system within the private network of the customer. The quantitative evaluation showed the overheads introduced by the Contract Validation function paled in comparison to those introduced by the MITM. Despite the cryptographic overheads of the MITM, the End-to-End connection times show the architecture is suitable to common IoT applications. Nevertheless, highly interactive applications, such as cloud-assisted Augmented Reality experiences, would likely face difficulties due to the near 40 ms of added delay (against the times measured without our solution).

V. Conclusion

We have successfully designed an IoT messages verification architecture which, using standard API description models, allows to create and enforce message contracts between the IoT solution provider and its customer. The use of existing libraries (flex), coupled with a custom-made whitelisting method, and a custom proxy wrapper that turned the validator into an NF, allowed to demonstrate the concept experimentally. The PoC prevented some abuses, such as data-leakage, attempted RCE exploitation (*EthernalBlue* vulnerability), and pivoting attacks (reaching targets in a private network otherwise inaccessible) through that API. The SFC-enabled MITM [5] handles the encrypted channels (HTTPS). The overall impact of the solution, when dealing with encrypted traffic, did not present a performance concern for typical IoT applications. However, low-latency applications such as Augmented Reality would likely be affected by the cryptographic overheads introduced by the MITM.

If the contract is changed, the verification process still works as long as the schema is updated. Additional data validation functions, tailored to specific purposes which would not be verifiable by the schema, are also possible in this architecture. The cloud computing capabilities of MEC-like systems, coupled with their traffic offloading capabilities, makes effortless the process of adding new functions. We have successfully presented a Security-aaS use-case using MEC, with the option of still deploying the NS locally. As long as we have trust in the fourth-party (the MEC operator), we can enforce scalable and on-demand perimeter defense using the Edge. Additionally, we may avoid significantly increasing the power consumption in the premises of the IoT equipment by offloading the service to the Edge, which is essential when the location lacks a connection to the power grid.

Acknowledgment

This work is supported by the European Regional Development Fund (FEDER), through the Regional Operational Programme of Lisbon (POR LISBOA 2020) and the Competitiveness and Internationalization Operational Programme (COMPETE 2020) of the Portugal 2020 framework [Project 5G with Nr. 024539 (POCI-01-0247-FEDER-024539)].

References

[1] I. Farris, T. Taleb, Y. Khettab, and J. S. Song, "A Survey on Emerging SDN and NFV Security Mechanisms for IoT Systems," *IEEE Communications Surveys Tutorials*, pp. 1–26, Aug 2018.

[2] A. Mihovska and M. Sarkar, *Smart Connectivity for Internet of Things (IoT) Applications*. Cham: Springer International Publishing, 2018, pp. 105–118.

[3] C. Stergiou, K. E. Psannis, B.-G. Kim, and B. Gupta, "Secure Integration of IoT and Cloud Computing," *Future Generation Computer Systems*, vol. 78, pp. 964 – 975, 2018.

[4] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate IoT Edge Computing with Lightweight Virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, Jan 2018.

[5] V. A. Cunha, M. Carvalho, D. Corujo, J. P. Barraca, D. Gomes, A. E. Schaeffer-Filho, C. R. P. D. Santos, L. Z. Granville, and R. L. Aguiar, "An SFC-enabled Approach for Processing SSL/TLS Encrypted Traffic in Future Enterprise Networks," in *2018 IEEE Symposium on Computers and Communications (ISCC 2018)*, Natal, Brazil, Jun. 2018.

[6] J. Wilson, R. S. Wahby, H. Corrigan-Gibbs, D. Boneh, P. Levis, and K. Winstein, "Trust but Verify: Auditing the Secure Internet of Things," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '17*. New York, New York, USA: ACM Press, 2017, pp. 464–474.

[7] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, "Multi-Context TLS (mcTLS)," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 199–212, 2015.

[8] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: Deep Packet Inspection over Encrypted Traffic," *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*, pp. 213–226, 2015.

[9] J. Han, S. Kim, J. Ha, and D. Han, "SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module," in *Proceedings of the First Asia-Pacific Workshop on Networking*, ser. APNet'17. New York, NY, USA: ACM, 2017, pp. 99–105.

[10] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A Free and Open Source Interactive HTTPS Proxy," 2010–. [Online]. Available: https://mitmproxy.org/

[11] V. Buterin, "Ethereum: A Next-generation Smart Contract and Decentralized Application Platform," accessed: 15-01-2019. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[12] ETSI ISG NFV, "Network Functions Virtualisation (NFV); Architectural Framework," *ETSI GS NFV 002 v1.2.1*, 2014.

[13] ——, "Network Functions Virtualisation (NFV); Management and Orchestration," *ETSI GS NFV-MAN 001 V1.1.1*, 2014.

[14] ETSI ISG MEC, "Mobile Edge Computing (MEC); Framework and Reference Architecture," *ETSI GS MEC 003 V1.1.1*, 2016.

[15] C. Parada, F. Fontes, C. Marques, V. Cunha, and C. Leitao, "Multi-Access Edge Computing: A 5G Technology," in *2018 European Conference on Networks and Communications (EuCNC)*. IEEE, 2018, pp. 277–9.

[16] J. M. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7665.txt

[17] F. Petitcolas, R. Anderson, and M. Kuhn, "Information Hiding-A Survey," *Proceedings of the IEEE*, vol. 87, no. 7, pp. 1062–1078, jul 1999.

[18] J. Wang, R. Zhu, and S. Liu, "A Differentially Private Unscented Kalman Filter for Streaming Data in IoT," *IEEE Access*, vol. 6, pp. 6487–6495, 2018.

[19] J. Fridrich, M. Goljan, and Rui Du, "Detecting LSB steganography in color, and gray-scale images," *IEEE Multimedia*, vol. 8, no. 4, pp. 22–28, 2001.

[20] L. Sha, F. Xiao, W. Chen, and J. Sun, "IIoT-SIDefender: Detecting and Defense against the Sensitive Information Leakage in Industry IoT," *World Wide Web*, vol. 21, no. 1, pp. 59–88, Jan 2018.

[21] M. Antunes, D. Gomes, and R. L. Aguiar, "Towards IoT Data Classification through Semantic Features," *Future Generation Computer Systems*, vol. 86, pp. 792–798, Sep 2018.

[22] P. Merriam, "Swagger Schema Validator," accessed: 15-01-2019. [Online]. Available: https://github.com/pipermerriam/flex