Anna van Buerenplein 1
2595 DA Den Haag
P.O. Box 96800
2509 JE The Hague
The Netherlands

www.tno.nl

T +31 88 866 00 00

Research on Networks (RoN) 2017
Programmable Data Plane

Date 13 February 2018

Author(s) Piotr Zuraniewski, Harm Schotanus, Jeffrey Panneman, Niels van Adrichem, Bart Gijsen, Robert Seepers (TNO), Adam Drescher (Washington University, Saint Louis)

Contents

1 Introduction ... 3

2 Use case description ... 4
2.1 Practical Setting ... 4
2.2 Trust Model ... 5

3 Offload Architecture ... 7
3.1 Signing ... 7
3.2 Encryption ... 7
3.3 NDN-IoT packet ... 8
3.4 Gatekeeper's operation ... 9

4 Selected data plane programming platform ... 12

5 NDN-IoT gatekeeper implementation ... 13

6 Validation ... 15
6.1 Testbed ... 15
6.2 "Smoke test" ... 15
6.3 NDN parsing, encryption and signing ... 16

7 Conclusions and next steps ... 17

8 References ... 19

1 Introduction

Since 2013 TNO, in collaboration with SURFnet, has been investigating SDN technology intended to create user instantiated network connectivity. The series of research projects have been centered around the CoCo (Community Connect) demonstrator that facilitates users to instantiate VPNs (without the assistance from a network administrator) that may span multiple network domains.

Complementary to controller-based programmable networking, data plane programmability is an emerging topic in the softwarized networks community. In 2016 a graduate student at SURFnet explored this research area by investigating packet authentication with P4 technology in a Mininet environment. In that year, TNO explored flexible protocol header parsing by exploiting eBPF (extended Berkley Packet Filter) technology that enables execution of simple packet manipulations for network interfaces.

In 2017, TNO and SURFnet decided to conduct a joint research project regarding programmable data plane. The following high-level research questions were guiding us in our activities:

- What is the **state of the art** in architecture, principles and available standards (in all their flavours, i.e., community standards, de facto standards, industry standards etc.) for the programmable data plane? What are the currently available (software and hardware) possibilities to implement data plane programmability. Which development tools are available to practically realize a proof-of-concept?
- **How can data plane programmability be used** to facilitate deployment of emerging technologies such as IoT and ICN? Can some inherent problems related to these new frameworks (e.g., poor security in IoT) be resolved using programmable data plane technology?
- What is the **performance of a proposed solution** and what are the ways to improve it? Specifically, is hardware acceleration possible? **What are the limits** in terms of scalability of simulation- and hardware-based (if applicable) environments where programmable data plane is to be deployed?

2 Use case description

In our research, we view network softwarization as an enabler for new services and means to facilitate deployment of other (disruptive) technologies. Specifically, we decided to investigate how data plane programmability can increase security of Internet of Things (IoT) devices. This research is focussed on IoT equipment that runs some flavour of the Information Centric Networks (ICN) protocol suite, which is a deployment choice advocated by many researchers and practitioners [1]. An interesting observation is that popular ICN implementations assume that security functions (data chunks signing, encryption) are built into the protocol. At the same time, however, most IoT devices cannot perform relatively expensive cryptographic operations due to CPU/memory/battery restrictions, making them unsuitable for secure communication at even modest sending rates.

In our research, we propose an ICN-IoT scheme which offloads the encryption of sensitive content to a trusted, more powerful machine called the *gatekeeper*. The gatekeeper exploits programmable data plane concepts by processing ICN datagrams to find if and which crypto operations need to be performed and then processes the packets accordingly. This solution is a first step towards providing robust encryption at reasonable speeds for IoT devices.

2.1 Practical Setting

In more detail we can illustrate the research problem and the proposed solution as follows.

Assume Alice owns a number of IoT devices that are connected on her own internal network. Further, consider each of her devices as trusted, i.e., they are assumed to be free of malicious intent. Alice's trusted internal network is connected to an untrusted external network through a single[1] gatekeeper, which takes care of interfacing between the two domains. Both the internal and the external network communicate using ICN[2]: Alice's devices are producers, while the devices on the external network are consumers that may express an interest in the produced data. Such a setup is represented in Figure 1.

[1] Can be logical, like in HSRP/VRRP setting but these considerations are outside of the scope of this document.

[2] In fact, the internal network employs a derived version of one of the ICN implementation (ICN-IoT) that, through a slight modification of the ICN data packet structure, is better suited for facilitating encryption offloading. The necessity for ICN-IoT data packets and its modified structure will become apparent in Section 3.

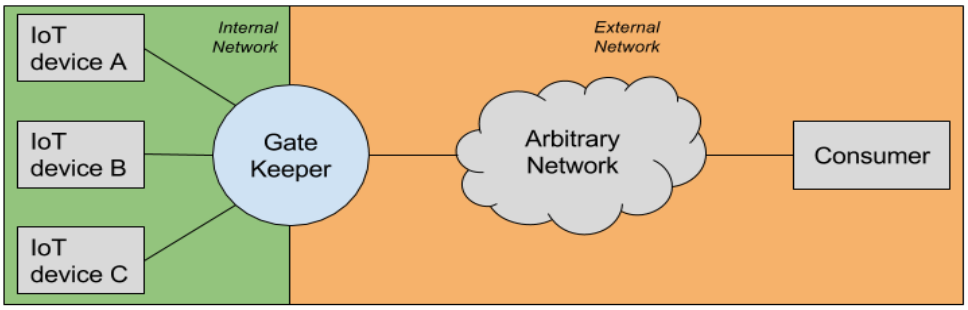

Figure 1: The logical relationship between devices in the Gatekeeper encryption offload scheme.

Alice decides that the data produced by her IoT devices is relatively valuable, and she will only share this data with consumers that she trusts and in a secure manner. To exchange data securely it has to be both encrypted for confidentiality purposes, and signed using a digital signature to guarantee data integrity and authenticity. Unfortunately, Alice's IoT devices are relatively weak and are unable to perform the computationally expensive cryptographic operations involved. Facing such limitations, Alice wishes to delegate the cryptographic workload to the gatekeeper, which is substantially more powerful than an IoT device and may, thus, encrypt and sign the produced data on the IoT device's behalf.

For the most part, the gatekeeper behaves as a regular ICN router. When a consumer expresses an interest in one of the IoT device's data, the gatekeeper will first identify if the interest matches a packet in its content store, as is done in regular ICN. If the encrypted data is not yet available, it forwards the interest to the internal network. The IoT device will respond to this interest by sending the unencrypted data to the gatekeeper. The gatekeeper encrypts this data using the cryptographic key for that IoT device and then signs the resulting ICN packet on behalf of the IoT device. For the purposes of this paper we assume that the keys for the producers, consumers, and gatekeeper were all preconfigured. This data may subsequently be shared over the external network in a secure manner: decryption is only possible by consumers who have the key for that IoT device.

2.2 Trust Model

Given that the IoT devices do not have the resources to perform cryptographic workloads, it is impossible to facilitate secure data exchange without making some assumptions of trust within a part of the network. In this work, we assume there exists an *internal network* -- a trusted domain in which all devices are assumed to be trusted and free of malicious intent. This internal network consists of Alice's IoT devices and the gatekeeper's interfaces that are not pointing towards an external (untrusted) network, as denoted by the bounding boxes in Figure 1. While this may appear as a strong assumption for the general case, we consider it acceptable for an environment in which the IoT devices are isolated from the external network. To achieve such isolation, one might: physically prevent adversaries from connecting to the internal network, e.g., security gates in a building; require proximity between the devices in the internal network; or, use out-of-band communication. This assumption is not significantly different from most existing security architectures, in which the security

of a network is not warranted after an adversary obtains access to the internal network.

A second assumption made in this work, though not directly pertaining to encryption offloading itself, is that all trusted consumers trusted by Alice are allowed to obtain the data produced by *all* of Alice's devices. This is a simplification of a more generic case in which Alice may only allow certain consumer-producer pairs to exchange data. The generic case would require the gatekeeper to maintain multiple keys, one for each producer-consumer pair, and brings forth the issue of key management. In our simplified scenario, it is possible to encrypt all data produced by Alice's devices with a single key that is stored on the gatekeeper and is shared by all consumers. This enables us to study the effects of encryption offloading in a vacuum, without considering the architectural modifications required by a gatekeeper and the required key-exchange or agreement protocols in ICN. In support of this simplified scenario, we also assume that the key is pre-distributed and can directly be used by the gatekeeper for encryption purposes. Note, however, that our architecture itself does not prevent more advanced scenarios that employ multiple key pairs.

3 Offload Architecture

In this Section we will discuss the encryption offload architecture in more detail. To be more specific in our considerations, we will pick Named Data Networking (NDN, [2]) implementation of ICN. At the same time, there are no fundamental obstacles in selecting a different flavour, provided that it is extensible (allows for “experimenter” TLV fields definition).

3.1 Signing

Let us first consider the case of data packet signing. The NDN stack mandates that all data packets must be signed by the producer [3]. The IoT devices considered in this work are not capable of performing this (cryptographic) signature operation. Therefore, it is not possible for them to employ (a generic) NDN and it is required to run some sort of a modified stack without signature-related fields. In this situation IoT devices will offload the signing operation to the gatekeeper.

If IoT devices would be able to sign packets themselves, then the signatures only have meaning within the internal (trusted) network. The reason is that the encryption operation will change the content of the packet which renders the original signature invalid, thus forcing the gatekeeper to calculate a new signature anyway. Besides, one can imagine that a direct communication of the external world with a particular IoT device may not be desired and in fact the gatekeeper will act as a proxy. Therefore, assigning the packets signing function to the gatekeeper makes a perfect sense from an architectural point of view.

3.2 Encryption

In our architecture we assume that encryption is an optional feature and that, if needed, IoT devices request this feature explicitly and in-band. There are a number of potential modifications of NDN that allow a producer to signal the gatekeeper that a packet should be encrypted. For example, by including an additional MetaInfo field that initiates the creation of a new NDN-packet type and naming conventions. We propose a slight modification of NDN to facilitate such signalling by introducing a new *EncryptMe* TLV field to a data packet.

3.3 NDN-IoT packet

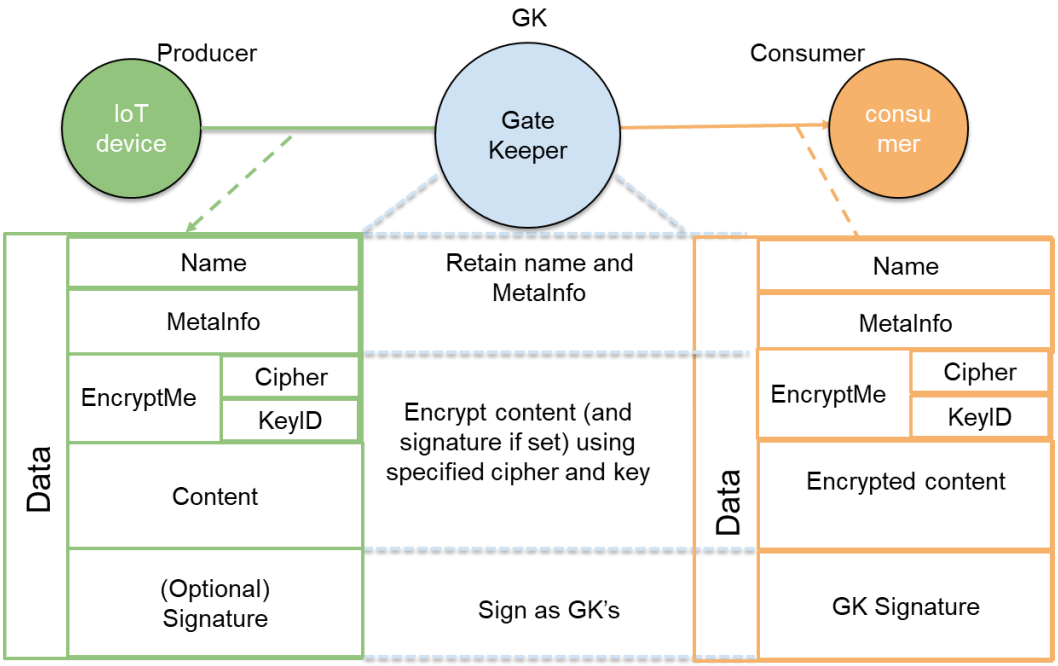

Figure 2: The gatekeeper processing an NDN-IOT data packet (left) into a signed NDN data packet with encrypted content (right)[3]

Figure 2 depicts (a) the modified version of the NDN-IoT data packet, (b) the operations performed by the gatekeeper to produce an encrypted and signed NDN packet as well as (c) the encrypted packet itself. Figure 3 and Figure 4 show the *wireshark* packet dump before and after encryption and with the old and the new signature.

It can be observed that the NDN-IoT data packet differs from a conventional NDN packet in two ways:

1. An optional new field of the type *EncryptMe* is inserted in a regular NDN packet as a sub-field of the Data TLV field. EncryptMe is a (possibly) nested TLV field, with type number of 33000 ("application use", see [4]) which presence specifies that the packet needs to be encrypted. This TLV may either have: (i) a length of 0, meaning that it is up to the gatekeeper to decide what key and cipher are to-be used for encryption; or (ii) children TLVs which are used by the producer to describe which cipher (type number of 33001) and key (type number of 33002) should be used for encryption.
2. The *Signature* field, a TLV that is used in NDN for verifying data integrity and provenance, has been made optional. This adaptation allows constrainted producers to send their data in an NDN-like fashion, while (slightly) more powerful producers may sign the packet (as is required in

[3] The interface could be generalized even further: the IoT devices might speak their own protocol, which is converted in the gate keeper. Feasibility of such a solution would be an interesting future work, illustrating what can and what cannot be done with data plane programming at this point in time.

regular NDN). Note that without a signature, it is impossible to validate the origin and integrity of a data packet. As our work assumes a full trust model for the internal network (as described in Section 2.2), the producers operating therein may abstain from signing their packets.

3.4 Gatekeeper's operation

As depicted in Figure 2, the gatekeeper processes the NDN-IoT data packet as follows:

(i) it checks if the *EncryptMe* field is present (that already may need a processing first up to Layer 4 if NDN packets are encapsulated as UDP payload and then processing nested TLVs of NDN packet itself). If so, then the gatekeeper administrates that the packet content should be encrypted and, if the header is of non-zero length, it obtains the preferred cipher and key identifier for encryption. The *EncryptMe* TLV remains untouched as it carries the information which will be used for decryption;

(ii) if the condition of (i) holds, i.e., if *EncryptMe* field is present then the packet content field (i.e., full TLV field, not only the Value of Content TLV) is encrypted using the specified key and cipher and the result of this operation is packed into the new TLV field, i.e., "Encrypted content" (type number of 33003) and the original "Content" is removed;

(iii) the *Name* and *MetaInfo* fields are copied into the NDN data packet;

(iv) the gatekeeper signs the, now constructed, NDN packet. If there was a signature of the sending IoT device present, it is simply discarded.

For decryption, the receiving party needs to perform operation (i) and the operation reverse to (ii) needs to be executed.

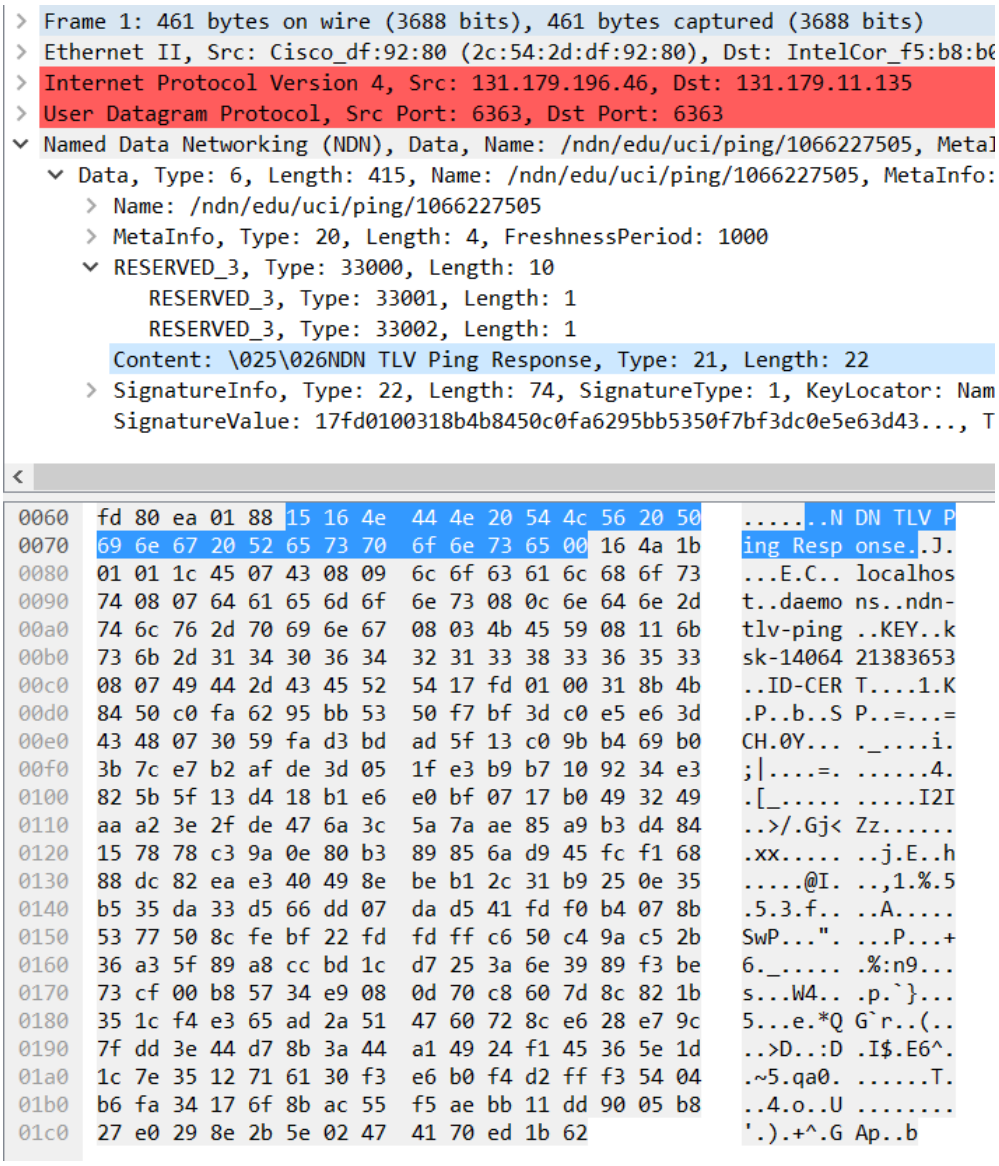

Figure 3: NDN packet before encryption. Custom EncryptMe TLV is visible (type=33000) along with two children TLVs (cipher and keyID)

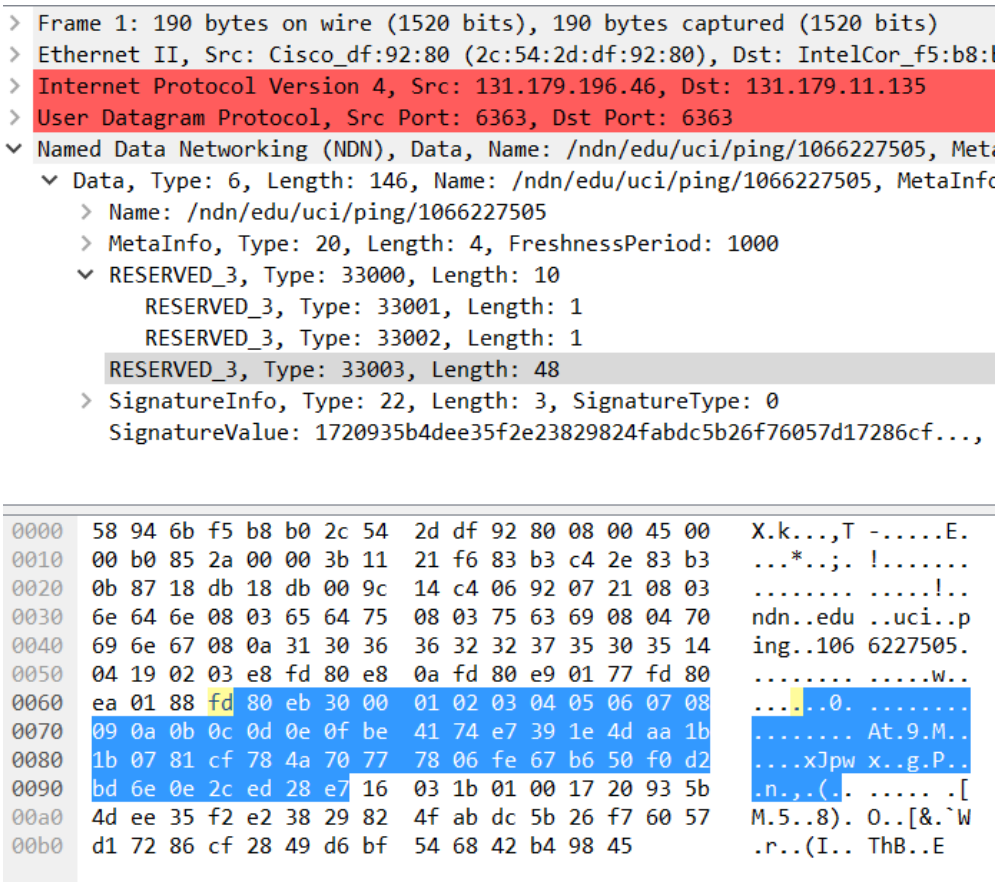

Figure 4: NDN packet after encryption. TLV with type=33003 is the encrypted content (note the presence of the initialisation vector 0x00 0x01...0x0f for the crypto algorithm). Gatekeeper signature is also visible and is much shorter than the original one due to usage of the different algorithm.

4 Selected data plane programming platform

The architecture described in Section 3 can be implemented on any general purpose server. However, the aim of this project was to explore the abilities of programmable hardware data plane platforms and its ability to achieve high processing speeds. At the beginning of 2017 when the project started the choice for the platform was straightforward. In fact, only Netronome's platform offered hardware accelerated cryptography and programmability in modern high-level languages along with the software development kit. All the developments have been done using Netronome AgilioLX platform which came with SDK v6 [5].

This platform offers three programming modes P4, microC and microcode. These modes can be mixed, for example, the P4 program can call microC routines. While P4 is target independent (i.e., the code is portable), the remaining two modes are Netronome specific. The microC language is a derivative of the language C. It is restricted in supported data types and operations (i.e., no floating-point) and it is extended by enabling the use of card specific architecture (e.g., hierarchical memory). The microcode is a low-level, assembly like language which is also platform-specific.

The LX card (as opposed to CX version) hosts "crypto engines", i.e., hardware elements capable to accelerate cryptographic operations. According to the manual, [p. 264] [6] *"NFP-6xxx includes a cryptography block includes 50Gbps cryptography block that supports Bulk cryptography plus authentication. (…) The bulk cryptography engines can run the following: AES, DES, 3DES, 3GPP UEA2(Snow3g f8) stream cipher, 3GPP UEA1(Kasumi f8) stream cipher, UIA2 Snow3G f9 (3GPP MAC), UIA1 Kasumi f9 (3GPP MAC) SHA-1, SHA-2, MD5 and ARC4."*

In the SDK one can find an example of an IPSec implementation, heavily relying on the microcode. We have not evaluated it because it was out of scope of this project. Unfortunately, we have concluded (and confirmed with the vendor) that it is not straightforward to access an arbitrary crypto function from P4/C level. The library that allows for that is planned to be released in 2018. The proposed workaround is described in Section 5.

5 NDN-IoT gatekeeper implementation

The high-level flowchart of our program is presented in Figure 5. In our work we assumed the use of the NDN protocol and its datagrams are encapsulated as UDP payload, which is further encapsulated in IP and Ethernet. The matching conditions up to Layer 4 are checked by P4 program (orange blocks). However, for processing of the NDN packet as well as executing the cryptographic functions we used microC (blue blocks). The reasons for this choice are following: NDN packets are built around the concept of a nested Type-Length-Value (TLV) field, i.e., the Value field may contain further TLVs and each of them may contain further TLVs and so on. Furthermore, not only the Value field, but also Type and Length are of variable length. The processing of such a complex structure proved to be very difficult in the P4 language, at least in the version 14 which was available at the time of performing this research, see [7] for the details. Moreover, P4 did not offer any cryptographic functions. To circumvent this difficulty, we decided to use the publicly available C implementation of a selected crypto function and tailored it to Netronome's microC requirements.

The encryption of the content of NDN packets causes some extra difficulty if the encryption algorithm makes the encrypted content larger than the original content field. The reasons for that may be padding (i.e., working only with the blocks of fixed size) and or/adding some extra information like the initialization vector. In NDN the packet Length field of any TLV is also of variable length. If the Value field has a size less than 253B, then the Length field has a size of 1B and stores the length of the Value field. However, for larger Value fields (from 254B to 65535B) the first octet of the Length field is fixed to 253 and two following octets code the actual length of the Value field, which makes the size of the Length field being 3B (even larger Value fields of 5B and 9B are also possible, see [8]). In the case where after encryption a border of 253B is crossed, the structure of Length field needs to be altered. To make the things even more complicated, we have to check if making a given child TLV larger has an influence on the size of its parent TLV. This makes it infeasible to construct an NDN packet in a sequential way, i.e. it is not possible to produce the data fields of an NDN packet in some predefined order.

At the same time we need to realize that the packet may also experience shrinking. For example, assume that a producer signed the packet and used a signing algorithm which provided a long key locator while the gatekeeper drops the whole original signature field (these are two child TLVs of DATA TLV) and puts a simple hash instead. That may cause a situation where the Length field which originally was encoded in 3B needs to be encoded in 1B.

These considerations prove that while nested TLV schema offers great flexibility, it comes with the price of high complexity of processing. This is therefore a very good use case for exploring the boundaries of programmable date plane technology.

During the project we succeeded in producing the code to run the NDN-IoT functionality on the selected target. The code has been published on TNO's Github

https://github.com/TNODigitalInnovations/p4_ndn_crypto

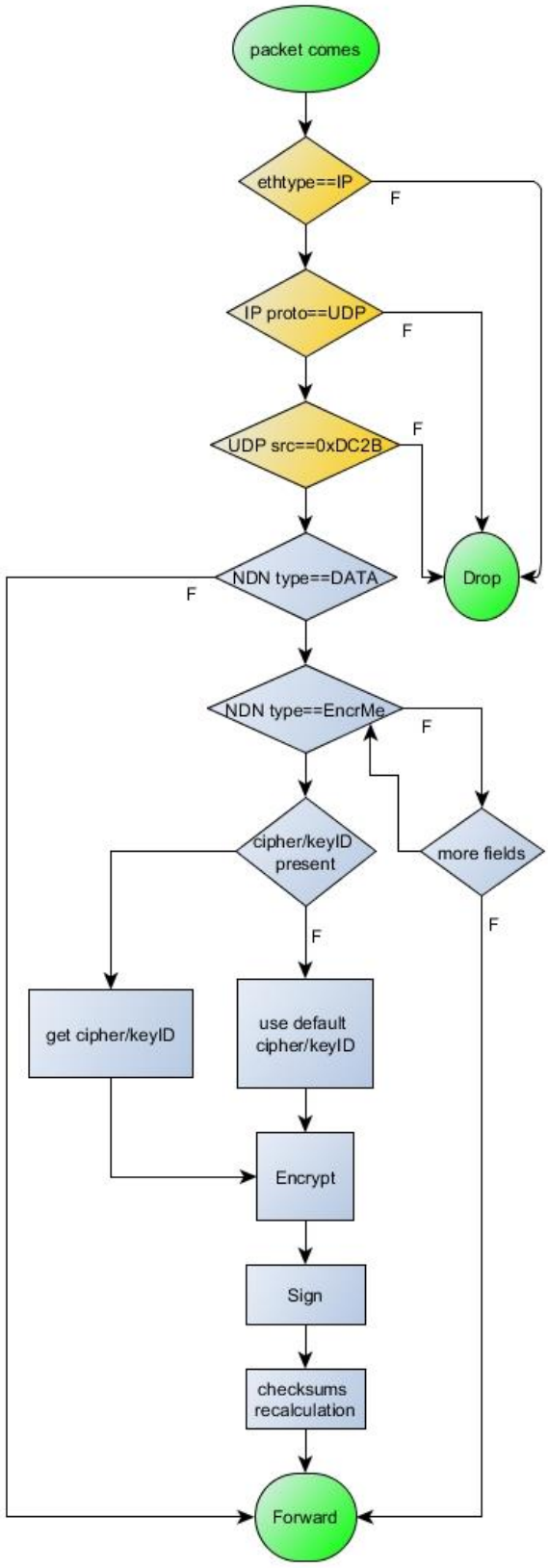

Figure 5: NDN packet processing flowchart. Orange blocks are implemented in P4 while the blue blocks are implemented in microC

6 Validation

6.1 Testbed

We have tested our implementation using a simple testbed depicted in Figure 6 which consisted of the following elements:

- Spirent STC-1 – hardware traffic generator, which sent NDN datagrams with "EncryptMe" TLV by replaying a crafted *pcap* file. It also served as a receiver/recorder of traffic returning after encryption. Each of its 12 ports can operate at 10Gbps.
- NoviFlow – hardware accelerated OF1.5+ SDN switch (16x10Gbps), used only to control traffic forwarding between the ports.
- Server with Netronome AgilioLX network interface card – running our encryption code and sending output via its another interface.
    - Card worked in 4x10G mode, using one break-out cable and one physical port.

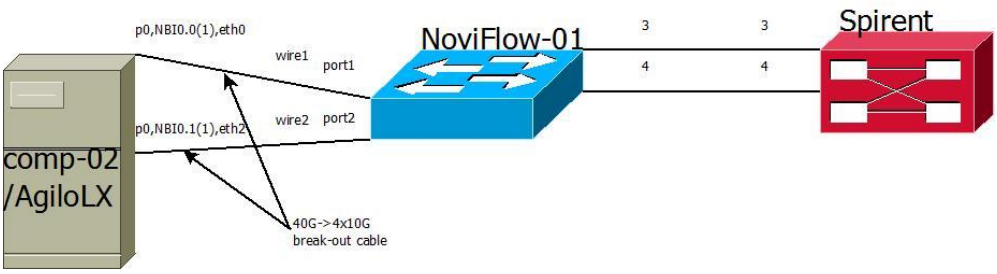

Figure 6: Testbed for validation

We have executed two simple performance evaluation tests which we describe in more details below.

6.2 "Smoke test"

The aim of this test was to verify the maximum switching capabilities of the card without performing any other operations. We have compiled a very simple P4 application (based on Netronome code) which made AgilioLX forwarding any traffic from one physical port to another. The card was put in 4x10Gbps mode and the traffic consisting on 64B UDP datagrams was sent at 10Gbps rate by Spirent.

Three main observations from the smoke test are:

1. About 2.5% of traffic was lost (see Figure 7). We have checked in a separate test that the NoviFlow switch was not a bottleneck (after short-circuiting ports 3 and 4 on the NoviFlow switch we saw 10Gbps as a received rate on Spirent port 4).
2. We saw some problems when the firmware was loaded while the card was already receiving the traffic. This case requires further investigation.
3. Debug mode of both the program and the Run Time Environment has a very heavy impact and needs to be turned off for any reasonable performance

test. Turning the debug on reduced the traffic rate from ~10Gbps to ~250Mbps.

Port Name	Tx L1 Rate (bps)	Rx L1 Rate (bps)	Tx L1 Rate (Percent)	Rx L1 Rate (Percent)
Port //1/3	10,000,000,059	0	100	0
Port //1/4	0	9,766,042,005	0	97.66

Figure 7: "Smoke test" results

6.3 NDN parsing, encryption and signing

In this test we used the software described in Section 5. As mentioned earlier, the application did **not** use hardware acceleration for cryptographic operations. Furthermore, the card was not optimized for performance, rather, the compiler optimization for size was needed, otherwise program was too large to be loaded onto the card. Moreover, the number of threads needed to be reduced, otherwise the program instruction limit was hit and the program did not compile. Due to the time constraints, no exploration of the card parallel programming model was done (switching between the contexts, usage of neighbouring cores etc.). Finally, for the unknown reason, we were not able to run in non-debug mode which in case of a “smoke test” had a huge impact (250Mbps vs. 10Gbps). The achieved rate was in the order of 2.7Mbps (Figure 8) which obviously is not a “production ready” speed.

Port Name	Tx L1 Rate (bps)	Rx L1 Rate (bps)	Tx L1 Rate (Percent)	Rx L1 Rate (Percent)
Port //1/3	9,999,580	0	0.1	0
Port //1/4	0	2,767,967	0	0.028

Figure 8: Test results for NDN parsing, encrypting and signing

7 Conclusions and next steps

The research reported in this report was conducted to explore the state of the art w.r.t. programmable data plane technology. Further, using an NDN-IoT use case we have explored if we can use the technology for developing new use cases and whether this can be applied at operational network performance levels.

At the start of this research we selected the only available hardware programmable data plane platform that supports the P4 programming standard: a Netronome AgilioLX network interface card. In itself this is already an indication that programmable data plane technology is in its infancy, yet.

To explore the capabilities of this equipment we designed the NDN-IoT scheme, that enables non-crypto-capable IoT devices to communicate in a secure way via an NDN gatekeeper. This gatekeeper communicates to the IoT devices in a trusted, internal NDN network and provides encryption functionality on behalf of the IoT devices towards others end points over untrusted, external networks. The nested TLV nature of NDN and the computationally complex encryption operations make the gatekeeper a secure-communication protocol converter that is far more complex than current data plane gateway solutions. As such, the ability to successfully develop and run a data plane gatekeeper demonstrates that programmable data plane technology does have the potential to innovate network services.

In this experiment we explored the limitations of our selected data plane programming platform, when applying it as the NDN-IoT gatekeeper. From this experiment we can conclude:

- We were able to program the Netronome card to execute the designed NDN-IoT scheme. However, we were not able to do so by using the standard P4 programming language. The complexity of the gatekeeper code and the lack of a cryptographic library were the primary reasons why we had to resort to using the Netronome proprietary programming language. Still, even in this mode we were not able to access hardware crypto “engines” (processors) and used the general-purpose ones. The vendor received our feedback and plans to enhance the accessibility of the crypto functions in 2018.
- Moreover, run-time constraints w.r.t. size of the executable code and the complier instruction limit prohibited us to fine-tune the developed gatekeeper code. As a result, the demonstrated gatekeeper is not able to capable to operate at ‘production ready’ traffic speeds.

Overall we can conclude that programmable data plane technology provides a promising outlook, especially to support further innovations in the fields of IoT, ICN and NFV. Admittedly, our explorations have demonstrated that the state-of-the-art still needs to make big steps towards operational maturity. Also, the learning curve prior to reaping the benefits of this technology are significant.

We plan to continue our research in a field of programmable networks. Several possible paths are:

- Use Netronome crypto library when available
- Learn more about optimization/profiling

- Measure performance
- Explore other topics and/or platforms
    - Security/network monitoring
    - VPP, Mellanox BlueField, Broadcom SDK

8 References

[1] Y. Zhang, “ICN based Architecture for IoT,” IETF, 16 07 2017. [Online]. Available: https://tools.ietf.org/id/draft-zhang-icnrg-icniot-architecture-01.html.

[2] “Named Data Networking,” [Online]. Available: http://named-data.net/.

[3] “NDN Packet Format Specification 0.2-2 documentation - Signature,” [Online]. Available: https://named-data.net/doc/ndn-tlv/signature.html.

[4] “TLV-TYPE number assignment,” [Online]. Available: https://named-data.net/doc/ndn-tlv/types.html.

[5] “Agilio LX SmartNICs,” [Online]. Available: https://www.netronome.com/products/agilio-lx/.

[6] *Netronome Network Flow Processor 6xxx. Flow Processor Core Programmer's Reference Manual,* 2017.

[7] S. Signorello, R. State, J. Franagois and F. O., “Ndn.p4: Programming information-centric data-planes.,” in *NetSoft Conference and Workshops*, 2016.

[8] “Type-Length-Value (TLV) Encoding,” [Online]. Available: https://named-data.net/doc/ndn-tlv/tlv.html.