A Type-Theoretic Model on NDN-TLV Encoding

Xinyu Ma
University of California, Los Angeles
Los Angeles, California, USA
xinyu.ma@cs.ucla.edu

Alexander Afanasyev
Florida International University
Miami, Florida, USA
aa@cs.fiu.edu

Lixia Zhang
University of California, Los Angeles
Los Angeles, California, USA
lixia@cs.ucla.edu

ABSTRACT

In Named-Data Networking (NDN), all packets are encoded in the Type-Length-Value (TLV) format. TLV encoding and decoding are implemented in every NDN library, and used by all applications and forwarders. Therefore, formal analysis of TLV encoding can assist NDN software development in the simplification of the code base, analysis of the performance, and improvement of robustness.

In this paper, we want to bring attention to the subtleties of TLV encoding. As an initial result, we develop a type-theoretical model of TLV encodable types, and give an algorithm to automatically derive encoding and decoding functions. We formally prove that the derived encoding and decoding functions are inverse to each other. To evaluate the practicality of automatically derived algorithms, we implement the proposed algorithms in C++ templates and evaluate them in three aspects: performance, memory usage, and code complexity. Our results show that our C++ library is competitive in these three aspects. Though our implementation is not fully automated, we show that it is possible to have a fully automated library in future that correctly produce the encoding and decoding functions. We also discussed the limitations of our model and problems worth attention. We hope our work can offer a starting point of further research on TLV, especially formal analysis and automated implementation.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; • **Networks → Presentation protocols**; *Network performance analysis.*

KEYWORDS

Named data networking, Information-centric networking, Encoding, Type-Length-Value format

ACM Reference Format:
Xinyu Ma, Alexander Afanasyev, and Lixia Zhang. 2022. A Type-Theoretic Model on NDN-TLV Encoding. In *9th ACM Conference on Information-Centric Networking (ICN '22), September 19–21, 2022, Osaka, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3517212.3558093

1 INTRODUCTION

The Type-Length-Value (TLV) encoding allows fields to have variable sizes, and has been widely used in network protocols. As a new network architecture, Named Data Networking (NDN) also defines its network protocol in the TLV format. The implementation of TLV encoding and decoding becomes essential to every piece of NDN software. NDN is still under active development, some of the research activities focus on the performance of the forwarding pipeline [7, 20, 28], and the implementation of NDN forwarders [3, 14, 18]. Since the packet format of NDN has a potential to change over time, there would be a need to evolve the encoding implementations. Different from fixed-length encoding, the encoding and decoding procedure is more complex. Therefore, it is important to pay attention to the subtleties of TLV encoding, such as the formal model and the implementation of TLV.

There are mainly three reasons why a model is useful: a) To save human effort. NDN is a new network protocol whose encoding format is actively evolving over time. A formal model can be used to automatically derive encoding and decoding functions, which makes the evolution of implementation easier when the protocol changes. Depending on specific programming languages, derived functions can be in different forms such as generic functions that handle all encodable types, templates or macros expanded at compile-time, and functions generated by some scripts. b) To improve software robustness. Bugs in the encoding/decoding code may lead to software vulnerabilities, as code that handles untrusted inputs is often exploited by attackers. A formal model enables formal verification and further helps in writing bug-free code. For example, OpenSSL has 205 vulnerabilities since 1999 [6], most of which are remotely exploitable. There are bugs (such as CVE-2016-2108) caused by the careless implementation of encoding code. Similar things probably happen to TLV encoding in NDN world. If we can prove the decoding code only parses legally encoded objects and does not crash on illegal inputs, the software will be more robust to attacks. c) To give a basis for performance improvement in encoding. Generally speaking, encoding is unlikely a performance bottleneck concern for a specific application. However, since it happens at every running application instance, the cumulated usage of computing resources can be significant [12]. A formal model can be used to analyze performance and improve implementation.

In this paper, we make a first step toward an NDN-TLV encoding model via the following contributions:

- We develop a basic type-theoretic model of encoding and decoding, which can be used to automatically derive encoding and decoding functions.
- We formally define the correctness via inverse relation, and use computer proof assistant F* [22] to prove the relation holds for our model. This improves robustness in the sense that programs obtained from the model will properly handle illegal inputs.

ICN '22, September 19–21, 2022, Osaka, Japan

ACM ISBN 978-1-4503-9257-0/22/09.
https://doi.org/10.1145/3517212.3558093

- We show that it is practical to automatically derive encoders and decoders from a type-theoretical model, and discuss the future work to be done towards it.
- We discuss some subtle issues that are worth attention when designing and developing NDN libraries.

This paper is structured as follows. § 2 gives background in NDN TLV encoding and monadic parsing. § 3 presents a type-theoretic model of TLV encoding, and defines and proves the inverse relation of encoding and decoding functions. § 4 discusses the limitations of the current model and other issues related to NDN-TLV. § 5 evaluates the practicality of having automatically derived encoders and decoders. § 6 introduces related works. § 7 concludes the paper with future work.

2 BACKGROUND

2.1 NDN Packets

Named Data Networking (NDN) [11] based on the lessons learned from using and upgrading current networking stacks, put forward an objective for future evolvability of the protocol [25]. In particular, this objective is realized through (1) the use of variable-length encoding of protocol messages: Interest and Data packets, and (2) the requirement to gracefully process of unrecognized "non-critical" elements and invalidate packets with unrecognized "critical" ones. In other words, these packets have no fixed-length fields, requiring basic structural parsing to access fields. Examples of Interest and Data packets and the elements that can appear in these packets are shown in Figure 1. In this example, the Interest requests a file named "`/file/a`", requesting forwarders to return Data exactly matching the specified name (*CanBePrefix* element) and keep Interest state while waiting for Data for specified lifetime (*InterestLifeTime* element). The corresponding Data packet carries "`/file/a`", meta information, actual content, as well as signature information and signature itself. [1]

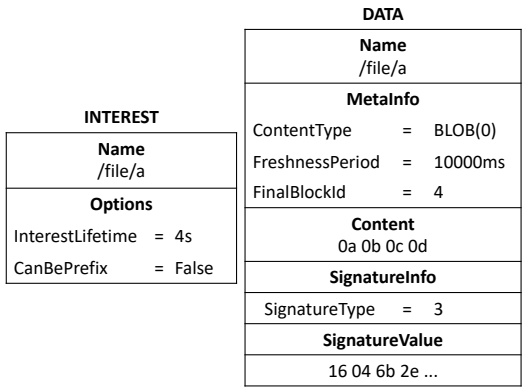

Figure 1: Example Interest and Data packets

2.2 TLV Encoding and NDN-TLV

The Type-Length-Value (TLV) encoding is a type of variable-length encoding. As indicated by the name, TLV puts a *type number* and the *length* before the value of an encoded object. The *type number* defines the type of this object, and the *length* indicates the length of its encoded value in bytes. Due to its flexibility, TLV encoding adopted by a wide range of network protocols, such as TLS extensions [17], BGP parameters [16], QUIC frames [10], and ASN.1 DER [9]. These protocols or formats typically have a fixed-length *type number* and *length* part, such as one byte or two bytes.

NDN also adopts a similar TLV encoding. However, To provide additional support for flexibility and protocol evolvability, NDN specification [24] chose to apply the following *variable-length* encoding to *type numbers* and *lengths*, which makes it more complicated that most existing TLV formats. If the value (of type number of length) is less than 252, the value is encoded in a single byte; otherwise, encoding consists of octet with values 253, 254, or 255 followed by 2-, 4-, or 8-byte encoded value, respectively.

For example, the Data packet in Figure 1 is encoded to Figure 2. In the figure, type names are **underscored**, and lengths are *italic*.

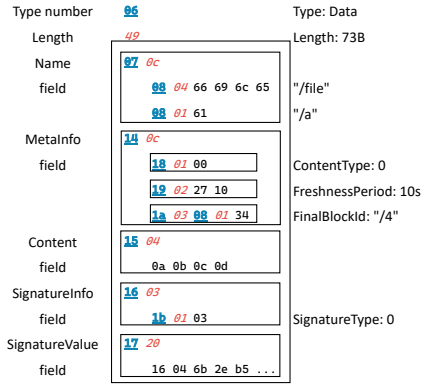

Figure 2: Encoded Data packet

2.3 F* and EverParse

Everest is a project focuses on security communication. The Everest team developed a general-purpose proof assistant F* [22], and presented EverParse [15], which is a framework that generates parsers and encoders for TLS messages from descriptions. The authors defined the correctness, safety, and non-malleability of parsers. They used F* to formally verify the generated parsers and encoders satisfy these properties. The authors proposed a domain-specific language (DSL) to describe TLS messages, implemented a compiler to convert DSL-defined structures into F* code. The output F* code is verified and derived into C code.

However, the TLV format used in TLS differs from NDN-TLS in two places: TLS-TLV has a fixed Type and Length field, but NDN-TLV has a variable-size encoding for them. TLS only uses TLV in some parts of the packet, but NDN-TLV uses it for every field. These differences make the DSL defined in EverParse not suitable for NDN-TLV. Our work takes a similar method in terms of using F* to verify the correctness of TLV encoding, but we focus on NDN-TLV and have not proposed a DSL yet.

There are other libraries designed for TLV encoding, discussed in § 6.

[1] The example is designed for demonstration purpose. A real NDN Data packet should be signed by a key and has a KeyLocator field carrying information about the key.

3 MODELING OF TLV

This section develops a basic type-theoretic model of TLV encoding, where encoders and decoders can be derived from the construction of encodable types. One benefit of a formally defined model is the ability to express the data structures in a declarative way so that the corresponding encoder and decoder can be automatically derived. Then, all a programmer has to do is to construct the encodable type, without the need to implement any algorithms manually. This reduces human workload and improves modularity. Another benefit is the formal verification of TLV encoding, which guarantees the correctness of algorithms and improves reliability.

This section is organized as follows. First, we make some definitions and assumptions used in the modeling. Then, we define encodable types — the input of encoding and output of decoding. After that, we briefly introduce the derived algorithms and prove the inverse relation (i.e. correctness) between encoding and decoding. At last, we make some discussions on this model.

3.1 Definitions

To be clear and precise, we define the terms and symbols used in the paper. Most of them have been widely used in programming language field (see [21]). We use $v : \texttt{t}$ to denote the value v is of type `t`. Let $f : \texttt{t1} \to \texttt{t2}$ denote the type of function f which takes an argument of `t1` and returns `t2`. All functions are supposed to be currying (i.e. a function taking two arguments of types `t1` and `t2` and returning `r` is written as $f : \texttt{t1} \to \texttt{t2} \to \texttt{r}$). We also assume that the following types and functors[2] are pre-defined:

- **`Uint64`** represents an unsigned 64-bit integer.
- **`Optional t`** represents an option value of type `t`, similar to OCaml's `'t optional` or Haskell's `Maybe t`. It has two legal constructors: **`None`** for the absence of a value, and **`Some`** v for the present of value v, with v of type `t`.
- `[t]` represents a list of type `t`, which also has two constructors: empty list `[]`, and concatenation `v::l`, with `v` of type `t` and `l` of type `[t]`.
- Unit type `()`, which only has one value `()`. This is used as a placeholder type when values are not required, as well as to construct boolean TLV fields.
- Product type `t1*t2`, whose legal values are pairs `(v1,v2)`, with `v1` of type `t1` and `v2` of type `t2`.
- **`ByteString`** represents a list of octets. We assume the type **`ByteString`** has the functions to obtain an empty byte string, get the length, take an arbitrary substring[3], and concatenate:

empty :**`ByteString`**

len :**`ByteString`** → **`Uint64`**

sub :**`ByteString`** → **`Uint64`** → **`Uint64`** → **`Optional ByteString`**

\+ :**`ByteString`** → **`ByteString`** → **`ByteString`**

In this paper, an *object* is something that can be encoded, such as a Data packet, an NDN name, the content part of some Data, and even a type number. The type of an object is called an **`EncodableType`**, which defines the legal values, the semantics, and the encoding format of a specific kind of object. We will formally define the universe of encodable types later. A *wire* is the result of encoding, which has the type **`ByteString`**. Roughly, we have the diagram

$$\textit{Object} : \texttt{EncodableType} \underset{\text{decode}}{\overset{\text{encode}}{\rightleftarrows}} \textit{Wire} : \texttt{ByteString}$$

However, decoding may fail when the input is illegal, and may consume part of the input wire. Therefore, in our work, we consider the two functions have the following type: assuming t is an encodable type,

$$\text{encode} : \texttt{t} \to \texttt{ByteString}$$

$$\text{decode} : \texttt{ByteString} \to \texttt{Optional}(\texttt{t} * \texttt{ByteString})$$

If decode fails, it returns **`None`**; otherwise, it returns some tuple of the parsed value (of type `t`) and the remaining wire (of type **`ByteString`**). Note that decode is a *total function*, which means it must be proved not to crash or fall into infinite loops on any input, not matter succeeds or fails.

Then, the inverse relationship between encoding and decoding can be expressed as follows:

$$\begin{aligned} \forall v : \texttt{t}, \quad & \text{decode}\ (\text{encode}\ v) \\ & = \texttt{Some}\ (v, \text{empty}) \qquad (1) \\ \forall v : \texttt{t}, \forall x, y : \texttt{ByteString}, \quad & \text{decode}\ x = \texttt{Some}\ (v, y) \\ & \Rightarrow x = \text{encode}\ v + y \qquad (2) \end{aligned}$$

A *TLV block* is the structure consisting of a type number, a length, and its value. A *field* is a TLV block as a part of an encodable type.

3.2 Encodable Types

We define encodable types via recursive construction as follows. Details are explained in the following subsections.

(1) (*Primitive types*) *TL number* (type for TLV-TYPE and TLV-LENGTH), *natural number* (type of unsigned 64-bit integer), *binary string*, *unit*, and *name* are encodable types.
(2) For each encodable type t and any natural number *type*, a *TLV block* with specific type number **`Tlv`**⟨*type*⟩ `t` is an encodable type. is an encodable type.
(3) For each type `t` obtained by (2), i.e. TLV blocks, the list type `[t]` is an encodable type.
(4) For each type t obtained by (2), the optional type **`Optionalt`** is an encodable type.
(5) For each type `t1,t2` obtained by (2), (3), (4), or (5), their product `t1*t2` is an encodable type.

3.2.1 Primitive types. We define 5 primitive types: *TL number*, *natural number*, *binary string*, *unit*, and *name*. We use **`TlNumber`**, **`Uint64`**, **`ByteString`**, **`()`**, and **`Name`** to denote them, respectively.

TL number is a type defined for TLV-TYPE and TLV-LENGTH fields, which is an unsigned 64-bit integer. Their encoding formats follow the variable size encoding defined in the specification [24]: if it is less than 253, use one byte containing its value; otherwise, use the first byte as a flag to indicate the length of this variable and the following 2, 4, or 8 bytes to encode its value. For example, if the value is 254, then the wire will be 253, 0, 254, as it cannot be contained in one byte.

[2] A functor is briefly a function of types, which consumes one type and produces another type, like C-array.
[3] We assume the substring function takes substring starting index and the length as input. It returns None when the input is out of range, and **`Some`**`[]` when the length is 0.

A *natural number* represents a 64-bit unsigned integer. As stated in the specification, it is contained in 1, 2, 4, or 8 bytes, using the least number which can contain it.

A *binary string* represents an arbitrary octet string. It is encoded as given.

The *unit* type is used to compose boolean. Since it has only one possible value `()` with no semantics, it is encoded with the empty string.

A *name* represents an NDN Name, which consists zero or more name components. A name component is a TLV block whose type is user-defined and unknown at compile time. A legal value of a name is a list of pairs `[(n,v)]`, where n : `TlNumber` and v : `ByteString`.

3.2.2 Functors. In our definition of encodable types, rules (2)(3)(4)(5) give four types of functors: *TLV block* of given type, arbitrary TLV block, *list*, and *optional*.

A *TLV block* encapsulates an encodable type `t` with a specific type number and a length. We use `Tlv`⟨*type*⟩ `t` to denote it. For example, generic Name component type can be written as

$$\mathit{GenericNameComponent} = \texttt{Tlv}\langle 8\rangle\ \texttt{ByteString}$$

The legal values of Tlv⟨*type*⟩ t are the legal values of t.

A *list* is a sequence of zero or more values of type t, where t must be a TLV block. Let $[t]$ denote the list type of t.

Optional indicates an optional field of type t, where t is a TLV block, denoted by Optional t. A legal value for Optional t can be either Some followed by a legal value of t or None. For example, the optional Content field in Data (a binary string with type number 21) can be written as

$$\mathit{ContentField} = \texttt{Optional Tlv}\langle 21\rangle\ \texttt{ByteString}$$

whose legal values include Some("abcd") and None. A boolean field can be represented by an optional TLV block of unit. For example, the CanBePrefix field in an Interest is

$$\mathit{CanBePrefixField} = \texttt{Optional Tlv}\langle 33\rangle\ ()$$

whose legal values include

$$\mathit{True} = \texttt{Some}\ () \qquad \mathit{False} = \texttt{None}$$

3.2.3 Product. A *product* is a fixed-length tuple containing a value for each of its field multiplicand types. In our work, multiplicands can be any of TLV blocks, lists, and Optionals (with mutually different type numbers), but not primitive types. For example, MetaInfo is a product of the corresponding field type of ContentType, FreshnessPeriod, and FinalBlockId. We use $*$ to denote a product.

$$\mathit{MetaInfo} = \mathit{ContentType} * \mathit{FreshnessPeriod} * \mathit{FinalBlockId}$$

To simplify discussion, we consider products are left associative, and then every product can be reduced to a product of two encodable types.

3.2.4 Algorithms. Since the encoded format of all types are documented in NDN packet format specification, the derivation of encoding and decoding algorithms is trivial. We put the pseudocode in the appendix A.

3.3 Proof Sketch of Inverse Relation

Using F*, we have formally proven the inverse relation between the encoder and decoder holds. In this subsection, we give a sketch of the proof. To show the equation (1) and (2) holds for every encodable type, we induct on the structure.

3.3.1 Primitive types. It is clear that (1) holds as the code branches of `decode` are one-to-one corresponding to the `encode`. For natural number, ByteString and unit, since the `decode` only succeeds when there is nothing left, (2) also holds. For TL number, if `decode` succeeds with $\mathit{decode}x = \texttt{Some}(v, y)$, then we have $x = z + y$ with z being a ByteString of length 1, 3, 5 or 9. It is trivial to show that $z = \mathit{encode}v$ case by case.

Note that TL number's encoding indicates its length, so it has the following length awareness property[4]:

$$\forall x : \texttt{ByteString}, \forall v : \texttt{t}, \texttt{t}.\mathit{decode}(\texttt{t}.\mathit{encode}\ v + x) = \texttt{Some}(v, x) \tag{3}$$

Since a name is a list of TLV blocks, the correctness is proven in the following subsections.

3.3.2 Tlv Block. Consider type `Tlv`⟨n⟩ `t`. Let v be a value of this type. By (1) and (3) of TL number, we have decode encode v = `Some`(v', empty) if `t`. decode `t`. encode v = `Some`(v', empty) for some v'. This holds by (1) of `t`. Thus, (1) holds for TLV Block.

Suppose decode x = `Some`(v, y). Then, by the decode algorithm, we have z s.t. $x = z + y$ and decode z = `Some`(v, empty). By (2) of `t`, encode $v = z$. Then, since (2) holds for TL numbers, we can deduct that (2) holds for `Tlv`⟨n⟩ `t`.

TLV block also encodes its length into the wire, so the length awareness property (3) holds as well. (`decode` line 6-7,10-11)

3.3.3 List. Induct on the length of the list. Clearly (1) and (2) hold for empty list. Suppose $v = v0 :: vl$. Then `[t]`. encode v = `t`. encode $v0$ + `[t]`. encode vl. Since t is some TLV block, by the length awareness (3),

$$\texttt{t}.\,\text{decode}(\texttt{[t]}.\,\text{encode}\ v) = \texttt{Some}(v0, \texttt{[t]}.\,\text{encode}\ vl)$$

By the definition of [t].decode,

$$\texttt{[t]}.\,\text{decode}(\texttt{[t]}.\,\text{encode}\ v) = \texttt{Some}(v0 + vl', \text{empty})$$

with vl' given by

$$\texttt{[t]}.\,\text{decode}(\texttt{[t]}.\,\text{encode}\ v) = \texttt{Some}(vl', \text{empty})$$

By inductive assumption, $vl' = vl$, so (1) holds.

Suppose

$$\texttt{[t]}.\,\text{decode}\ x = \texttt{Some}(v0 :: vl, y)$$

By (3) of t, x = `t`. encode $v0 + z$ for some z with

$$\texttt{[t]}.\,\text{decode}\ z = \texttt{Some}(vl, y)$$

By the inductive hypothesis, `[t]`. encode vl+$y = z$. Thus, `[t]`. encode v+ $y = x$, which proves (2).

3.3.4 Optional. Optional is similar to a list of length at most 1. Thus, (1) and (2) hold similarly.

[4] Called "strong prefix property" in EverParse [15]

3.3.5 Product. List and optional of TLV blocks satisfy a weak version of length awareness: suppose t, t' are two types from TLV block, list or optional that have different type numbers:

$$\forall v : \texttt{t}, \forall v' : \texttt{t'}, \texttt{t}.\,\text{decode}(\texttt{t}.\,\text{encode}\, v + \texttt{t'}.\,\text{encode}\, v') = \texttt{Some}(v, \texttt{t'}.\,\text{encode}\, v')$$

Therefore, by induction on the length, we can prove that the inverse relation holds for products.

Suppose $(v0, vl)$: `t0*tl`, with `tl` being the rest of product and they do not overlap in type numbers. By the weak length awareness,

$$\texttt{t0}.\,\text{decode}(\texttt{t0*tl}.\,\text{encode}(v0, vl)) = \text{Some}(v0, \texttt{tl}.\,\text{encode}\, vl)$$

Thus, by induction hypothesis, (1) holds. (2) holds directly from the definition of decode.

4 DISCUSSION

This section discusses the limitations of the model and possible future improvements. We believe these are also problems worth attention when developing NDN libraries.

4.1 Unrecognized Fields

In consideration of evolvability, NDN defines the least significant bit of TLV type number as the critical flag. When the parser meets an unrecognized field with a non-critical TLV type, the specification allows the parser to ignore the field and continue parsing. However, in our model, the product type fails in every unrecognized field regardless of its type number.

To correctly describe the subtype relationship among structures, we have to use a more accurate type to replace the coarse model built upon products. The fact that an object with unrecognized fields is encoded from an extended structure definition should be correctly shown. For example, an unrecognized field in a MetaInfo may come from a newer version of the protocol that extends the MetaInfo type with new fields. However, the equation (2) could fail if we omit the unrecognized fields.

4.2 Encoding of Signature

During encoding, a known signature is nothing more than a byte string. However, the signature value is computed based on the encoded wire of previous fields. That is, if we consider the whole packet as an encodable type, signing happens in the middle of encoding. In our model, we assume that the product type is left-associative, so we could encode the part before the SignatureValue field and then sign it. For example, in our model, the type of Data packet becomes

$$\mathit{Data} = \mathit{PreviousFields} * \mathit{SignatureValue}$$

Thus, it is possible to encode PreviousFields, compute signature, and finally encode the whole packet with signature. However, this is less efficient in practice. Also, separating signature and the previous fields is counter-intuitive. Future work is needed to find a solution to include signing as part of the encoding.

The ParametersSha256DigestComponent in the Name of an Interest is computed after signing, and our model is unable to handle it.

4.3 Applications versus Forwarders

Our model is a basic step towards formal modeling and does not distinguish the use cases of NDN client applications and forwarders. However, in practice, the differences between a forwarder and a client application are remarkable, and they may deserve different models to handle. A forwarder needs to constantly decode packets, but seldom does encoding. It also has high requirements on throughput, so the developer may need to avoid unnecessary copying at all costs. After decoding, the original wire is usually discarded by a client application, but a forwarder needs to keep it as packets generally need to be sent out as is. Moreover, a forwarder sometimes needs to modify some field of a packet, e.g. adjusting the HopLimit value, so indexing to the memory of sub-elements in the wire is a useful feature to a forwarder, but probably less so to an application. Whether these differences can lead to the need for different theoretic models needs further research.

4.4 Languages Describing TLV Format

If automatically derived encoders are adopted, the language used to the format of a TLV encodable type is important, since it will not only be read by human-beings but also by programs that derive the encoders. Roughly, the languages can be categorized into four families: metasyntax language such as ABNF [4], domain-specific language (DSL) like Protobuf, programming language features such as Go struct tags, and meta-TLV.

The NDN specification [24] uses ABNF to describe all types. ABNF is good for human to read, and there are some general-purpose parser generators such as Yacc, GNU Bison and Lark [19]. However, ABNF as a metasyntax language is too general, which makes it difficult to associate an ABNF non-terminal with a class/structure in programming language. As a result, these general-purpose parser generators either requires the programmer's input to handle grammar rules, or simply returns an abstract syntax tree. Thus, for every encodable type, extra work[5] is needed to implement a practical parser. Moreover, existing general-purpose parser generators offer little help in encoding implementation.

DSLs can be a good fit for both human and machine use. Especially, Protobuf has a very similar format as NDN-TLV does. PyNDN2 [27] uses Protobuf to describe TLV types used in Sync protocols, and the author wrote a customized encoder that encodes Protobuf message objects into TLV wires, and a decoder doing the inverse. Though the Protobuf library does not provide a formal proof on the correctness of the encoders and decoders, the language itself does not prevent us from constructing formally-verified encoders. The only issue remained is how to indicate the fields covered by the signature. In our opinion, a DSL with Protobuf-like syntax plus some feature that allows annotations of fields would be an ideal solution.

Some programming languages provide features that allow designing a limited DSL. For example, Python's descriptors [8] and Go's struct field tags [1] support annotations to fields, and these annotations can be accessed at run time. These language features

[5] It is hard to estimate the amount of work since currently there is no such implementation. But the amount of work would be proportion to the number of types we need, which makes it not a fully automated method.

work seamlessly with structures or classes definition in the language, but it is difficult to translate among different programming languages.

If we only consider machine usage without human-readability, a TLV-based metasyntax language could also be designed, which describes object types and fields in the form of TLV elements. In short, using TLV to describe TLV types. Compared with human-readable languages, TLV metasyntax is easier to be transmitted via networks and more friendly to software parsers.

5 PRACTICALITY OF AUTOMATIC DERIVATION

Ideally, the implementation should be automatically derived from the proof of inverse relation by a automated theorem prover, such as Coq or F*. However, our model is not mature enough. Also, our current proof in F* relies on dependent types, a feature not supported in most programming languages and preventing the conversion. To evaluate the practicality of automatic derivation, we implemented a proof-of-concept library in C++ templates. We structured our program the same way as the model in § 3, used C++ template to implement the six ways of constructing an encodable type, and tested the encoding and decoding of Interest and Data packets to verify its correctness. Though the template code is written by hand, the encoding and decoding functions of specific objects are generalized from templates, so it can be considered as a semi-automated derivation. The implementation details are put in § B.

Theorem provers do not support all languages. However, since both Coq and F* support conversion to C and most programming languages have foreign function interfaces (FFI) that can use C libraries, automatically derived code could be used in a wide range of languages. An alternative approach is to take a semi-automated derivation method as we did in C++. For example, one may use features such as macros in Rust and Lisp, reflection and emitting in some .Net language like C#. One may also write a script to generate programming code for encoding and decoding functions in the designated language. Therefore, we think automated verification does not significantly narrow down the choices of programming languages.

We also evaluated our C++ library by comparing it with other implementations, including ndn-cxx[23], python-ndn[26], and YaNFD[14]. Since our model is not mature yet to serve the need of a client library, and also the library code is not fully automated as expected, it is unfair to compare it with production-ready NDN libraries. Thus, we think the numbers should be taken as a qualitative result of the practicality of our method, and put the detailed numbers in Appendix § C. We compared the implementations in three aspects: performance, implementation complexity, and memory overhead. Our results showed that the semi-automatically derived code is comparative in these aspects, which shows the derivation is a practical direction to implement NDN client libraries.

6 RELATED WORK

Nail [2] defines a protocol grammar to define both the wire format and the internal object model of data. Nail defines stream transforms to capture protocol features such as variable-sized fields and check-sums. These features provide great flexibility and reduce programmers' effort for safely parsing and encoding data. However, Nail streams are weakly typed, which may increase the risk of runtime errors.

The Network Protocol Tool (NPT) [13] uses a typed representation system that describes the protocol format. NPT compiles the description into a Rust parser. In addition, the authors also developed the Augmented Packet Header Diagram language that is machine-readable and can be used in IETF RFC documents.

There are works on verified encoding and decoding for different formats. Narcissus [5] designed a type-safe combinator-based system that can automatically derive decoders and encoders from a formal specification. The authors used the interactive proof to verify the derived encoder and decoders are inverses of each other. They evaluated their work by implementing the headers of five protocols used in the TCP/IP stack. However, Narcissus does not have first-class support for TLV.

There are domain specific languages, such as Protobuf, that can be used to describe TLV object types and generate encoding and decoding code with a code generator. However, the Protobuf library's encoding format is different from NDN-TLV, and it does not give a formal proof of correctness.

7 CONCLUSION AND FUTURE WORK

We designed a formal model of NDN TLV encoding and decoding by formally defined encodable types in a recursive way. This model promises correctness by formally proving the inverse relation of encoding and decoding functions. To assess the potential of automatically deriving encoding functions from a formal model, we implemented a C++ library that is structured in the same way as the model, and evaluated its performance, memory overhead, and implementation complexity.

In the future, we plan to refine our model to naturally include signatures and unrecognized fields, and write a better computer-based proof that can be effectively used to derive the desired automatic derivation. We also consider proposing a DSL to describe NDN-TLV objects after a more mature model is obtained, if necessary.

ACKNOWLEDGMENTS

We appreciate Professor Todd Millstein for helping revising algorithms and proofs. We would like to thank the anonymous reviewers for their valuable comments which helped us improve the paper's quality. This work was supported in part by US National Science Foundation under awards 1719403, 2019085, and 2126148.

REFERENCES

[1] The Go Authors. 2022. The Go Programming Language Specification. https://go.dev/ref/spec#Struct_types

[2] Julian Bangert and Nickolai Zeldovich. 2014. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 615–628. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

[3] Sung Hyuk Byun, Jongseok Lee, Dong Myung Sul, and Namseok Ko. 2020. Multi-Worker NFD: An NFD-Compatible High-Speed NDN Forwarder. In *Proceedings of the 7th ACM Conference on Information-Centric Networking (ICN '20)*. Association for Computing Machinery, New York, NY, USA, 166–168. https://doi.org/10.1145/3405656.3420233

[4] Dave Crocker and Paul Overell. 2008. Augmented BNF for Syntax Specifications: ABNF. RFC 5234. https://doi.org/10.17487/RFC5234

[5] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proc. ACM Program. Lang.* 3, ICFP, Article 82 (July 2019), 29 pages. https://doi.org/10.1145/3341686

[6] CVE Details. 2021. Openssl : List of security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-217/Openssl.html

[7] Chavoosh Ghasemi, Hamed Yousefi, Kang G. Shin, and Beichuan Zhang. 2019. On the Granularity of Trie-Based Data Structures for Name Lookups and Updates. *IEEE/ACM Transactions on Networking* 27, 2 (2019), 777–789. https://doi.org/10.1109/TNET.2019.2901487

[8] Raymond Hettinger. 2022. Descriptor HowTo Guide. https://docs.python.org/3/howto/descriptor.html

[9] International Telecommunication Union. 2002. Information Technology — ASN.1 Encoding Rules — Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). ITU-T Recommendation X.690.

[10] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[11] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. 2009. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09)*.

[12] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[13] Stephen McQuistin, Vivian Band, Dejice Jacob, and Colin Perkins. 2020. Parsing Protocol Standards to Parse Standard Protocols. In *Proceedings of the Applied Networking Research Workshop (ANRW '20)*. Association for Computing Machinery, New York, NY, USA, 25–31. https://doi.org/10.1145/3404868.3406671

[14] Eric Newberry, Xinyu Ma, and Lixia Zhang. 2021. YaNFD: Yet Another Named Data Networking Forwarding Daemon. In *Proceedings of the 8th ACM Conference on Information-Centric Networking (ICN '21)*. Association for Computing Machinery, New York, NY, USA, 30–41. https://doi.org/10.1145/3460417.3482969

[15] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *USENIX Security*. USENIX. https://www.microsoft.com/en-us/research/publication/everparse/

[16] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. https://doi.org/10.17487/RFC4271

[17] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446

[18] Junxiao Shi, Davide Pesavento, and Lotfi Benmohamed. 2020. NDN-DPDK: NDN Forwarding at 100 Gbps on Commodity Hardware. In *Proceedings of the 7th ACM Conference on Information-Centric Networking (ICN '20)*. Association for Computing Machinery, New York, NY, USA, 30–40. https://doi.org/10.1145/3405656.3418715

[19] Erez Shinan. 2022. Welcome to Lark's documentation! https://lark-parser.readthedocs.io/en/latest/

[20] Won So, Ashok Narayanan, and David Oran. 2013. Named data networking on a router: Fast and DoS-resistant forwarding with hash tables. In *Architectures for Networking and Communications Systems*. 215–225. https://doi.org/10.1109/ANCS.2013.6665203

[21] Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1 (2000), 11–49. https://doi.org/10.1023/A:1010000313106

[22] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. *SIGPLAN Not.* 46, 9 (sep 2011), 266–278. https://doi.org/10.1145/2034574.2034811

[23] NDN Project Team. 2021. ndn-cxx: NDN C++ library with eXperimental eXtensions. https://github.com/named-data/ndn-cxx/

[24] NDN Project Team. 2021. NDN Packet Format Specification version 0.3. http://named-data.net/doc/NDN-packet-spec/current/index.html

[25] NDN Project Team. 2021. NDN Protocol Design Principles. https://named-data.net/project/ndn-design-principles/

[26] NDN Project Team. 2021. python-ndn documentation. https://python-ndn.readthedocs.io/en/latest/

[27] Jeff Thompson. 2021. PyNDN: A Named Data Networking client library with TLV wire format support in native Python. https://github.com/named-data/PyNDN2

[28] Haowei Yuan, Patrick Crowley, and Tian Song. 2017. Enhancing Scalable Name-Based Forwarding. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 60–69. https://doi.org/10.1109/ANCS.2017.16

A ALGORITHM PSEUDO-CODES

This appendix gives the pseudo code of encoding and decoding. The pseudo-code is written in a language close to OCaml and Haskell. We also uses well-known functions such as `map`, `foldl` and `do` notations, when they make sense canonically.

A.0.1 Encoding. Encoding of primitive types is trivial:

```
(* Natural number *)
encode: Uint64 -> ByteString
encode v =
 (* Suppose UintX.encode gives the little endian
      encoding of an integer *)
 if v <= 0xff then Uint8.encode v
 else if v <= 0xffff then Uint16.encode v
 else if v <= 0xffffffff then Uint32.encode v
 else Uint64.encode v

(* TL number *)
encode: TlNumber -> ByteString
encode v =
 if v <= 252 then Uint8.encode v
 else if v <= 0xffff then
  Uint8.encode 253 + Uint16.encode v
 else if v <= 0xffffffff then
  Uint8.encode 254 + Uint32.encode v
 else
  Uint8.encode 255 + Uint64.encode v

(* ByteString *)
encode: ByteString -> ByteString
encode v = v

(* Unit *)
encode: () -> ByteString
encode () = empty
```

To encode a TLV block, we put its type number, length and encoded value in order

```
(* TLV block *)
encode: Tlv<n> t -> ByteString
encode v = let w = t.encode v in
 TlNumber.encode n + TlNumber.encode (len w) + w
```

Encoding of list is simply concatenating the encoding of its elements.

```
(* List *)
encode: [t] -> ByteString
encode l = foldl (+) empty (map t.encode l)
```

Encoding of an optional is the same as a single element list.

```
(* Optional *)
encode: Optional t -> ByteString
encode v = match v with
 | Some v0 -> t.encode v0
 | None -> empty
```

To derive the encoding of products, we consider products ($*$) and tuples ((`-`,`-`)) are left-associative. Then, every product is reduced to a product of two, and we encode them in order We simply concatenate

```
(* Product *)
encode: t1 * t2 -> ByteString
encode (v1, v2) = t1.encode v1 + t2.encode v2
```

A.0.2 Decoding. Decoding of primitive types is also trivial. For the sake of proof, only TL number consumes partial input. Decoders of natural number, ByteString and unit always try to greedily consume the whole input. This works because they are encapsulated in TLV blocks when used to compose another encodable type.

```
(* Natural number *)
```

```
decode: ByteString -> Optional(Uint64 * ByteString)
decode w =
let l = len w in
 if l = 1 then Some(Uint8.parse w, empty) else
 if l = 2 then Some(Uint16.parse w, empty) else
 if l = 4 then Some(Uint32.parse w, empty) else
 if l = 8 then Some(Uint64.parse w, empty) else
 None

(* TL number *)
decode: ByteString -> Optional(TlNumber*ByteString)
decode w = do
 (* <- means let w0 be the result if sub succeeds,
    and return None otherwise *)
 w0 <- sub w 0 1
 let v0 = Uint8.decode w0 in
 if v0 <= 252 then Some(v0, sub w 1 (len w - 1))
 else
  let l1 = 2**(v0-252) in
  w1 <- sub w 1 l1
  w2 <- sub w (1+l1) (len l - 1 - l1)
  (* Here we use the Uint64 decoding *)
  (v, _) <- Uint64.decode w1
  Some (v, w2)

(* ByteString *)
decode: ByteString->Optional(ByteString*ByteString)
decode w = Some (w, empty)

(* Unit *)
decode: ByteString -> Optional(() * ByteString)
decode w =
 if w = empty then Some((), empty) else None
```

The parser TLV block calls the parser of the type number, the length, and the value type in order.

```
(* TLV block *)
decode: ByteString->Optional(Tlv<n> t*ByteString)
decode w = do
 (n1, w1) <- TlNumber.decode w
 if n1 != n then None else
 (l, w2) <- TlNumber.decode w1
 wv <- sub w2 0 l
 (v, w3) <- t.decode wv
 if w3 != empty then None else
 w4 <- sub w2 l (len w2 - l)
 Some (v, w4)
```

The parser of the list $[t]$ repeats calling t's parser until it fails, and collect all results into a list:

```
(* List *)
decode: ByteString -> Optional([t] * ByteString)
decode w = match t.decode w with
 | Some (v, w1) ->
   let Some(vl, wl) = [t].decode w in Some(v::vl, wl)
 | None -> Some([], w)
```

Similarly, the decoder of $\mathtt{opt}(T)$ calls the parser of T, and encapsulates it with another Some.

```
(* Optional *)
decode: ByteString->Optional(Optional t*ByteString)
decode w = match t.decode w with
 | Some (v, w1) -> Some(Some v, w1)
 | None -> Some(None, w)
```

The decoder of a product calls the parser of its multiplicands in order.

```
(* Product *)
decode: ByteString->Optional((t1*t2)*ByteString)
decode w = do
 (v1, w1) <- t1.decode w
 (v2, w2) <- t2.decode w1
 Some ((v1, v2), w2)
```

B IMPLEMENTATION

This appendix discuss the implementation details of our proof-of-concept C++ template library. In B.1, we introduce how the users are supposed to use our library. In B.2, we summarize the implementation details that is not covered by the type theoretic model.

B.1 Overview

Our library allows users to use C++ structs/classes directly. The only thing users need to do is specify the format of the class as an encodable type. For example, the class of MetaInfo can be defined as Listing 1.

Listing 1: MetaInfo definition

```
struct MetaInfo {
  std::optional<uint64_t> contentType;
  std::optional<uint64_t> freshnessPeriod;
  std::optional<NameComponent> finalBlockId;

  using Encodable = Struct<MetaInfo,
  NaturalFieldOpt<0x18, MetaInfo, &MetaInfo::contentType
      >,
  NaturalFieldOpt<0x19, MetaInfo, &MetaInfo::
      freshnessPeriod>,
  NameComponentFieldOpt<0x1a, MetaInfo, &MetaInfo::
      finalBlockId>>;
};
```

The inner class, `Encodable`, contains the derived encoding and decoding function. It also has some data fields used in the encoding or decoding procedure. Listing 2 gives an example using this encoder class.

Listing 2: Using MetaInfo encoder

```
MetaInfo metainfo{
  .contentType = 0,
  .freshnessPeriod = 4000,
  .finalBlockId = GenericNameComponent(std::string("
      10000"))
};
// Encoding metainfo into buf
MetaInfo::Encodable encoder(metainfo);
Buffer buf(encoder.EncodeSize());
encoder.EncodeInto(buf.data(), buf.capacity());
// Decoding buf
const auto& [metainfo2, decoded_size] =
  MetaInfo::Encodable::Parse(buf);
```

B.2 Key Points in Design

B.2.1 Interfaces. The algorithm in § 3 relies on concatenation of byte string. However, this is an $O(n)$ operation, which is too slow to be used in practice. Therefore, in our implementation we pre-allocate a buffer before encoding, and let the encoder fill into the buffer. To allocate the encoding buffer, we need to obtain the size of the encoded wire before actual encoding. Therefore, the encoder interface is designed with three functions: `EncodeSize` calculates the size of wire; `EncodeInto` which encodes the object into an

allocated buffer; `Parse` parses the input wire returning the decoded object and size of wire consumed. See Listing 3.

Listing 3: Encoder interface

```
template<typename T, typename E>
concept Encodes = requires(T v, E e, uint8_t* buf,
    size_t len, const Buffer& wire) {
  E(v);
  {e.EncodeSize()}->std::convertible_to<size_t>;
  {e.EncodeInto(buf,len)}
    ->std::convertible_to<size_t>;
  {E::Parse(wire)}->std::convertible_to<std::tuple<
    std::optional<T>, size_t>>;
};
```

B.2.2 Product. Following the model in Section 3, primitive types are simply classes. C++ templates can be used to implement functors easily. The remaining problem is how to implement *products*, since we need to encode from a C++ class. C++ does not support reflection at the language level. Therefore, we need users' input to specify the fields of the class, which is possible via variadic templates (Listing 4). For each field, the encoder store its TLV type number and the offset in the class. Then, the value of this field can be obtained or set via the pointer to the object of this class.

Listing 4: Encoder of products

```
template<typename Model, typename ...Fields>
struct Struct {
  std::tuple<Fields...> fields;
  template<std::size_t I = 0>
  inline typename std::enable_if<I ==
      sizeof...(Fields), size_t>::type
  EncodeSize() const {
    return 0;
  }
  template<std::size_t I = 0>
  inline typename std::enable_if<I <
      sizeof...(Fields), size_t>::type
  EncodeSize() const {
    return std::get<I>(fields).EncodeSize() +
           EncodeSize<I+1>();
  }
  // Other functions omitted ...
};
```

B.2.3 Name component. Name components are not covered by the model, since they are TLV blocks but can have arbitrary type numbers, instead of a specific one known at compile time. The way we handle it is to define the name component class to be ByteString that contains the whole TLV block, and write specific encoding and decoding functions for it.

B.2.4 Signature. In § B.2.1, we calculate the encoded size before allocation. Here, a problem with signatures arises. As stated in 4.2, signature values cannot be computed before encoding, and sometimes even their lengths are unknown in advance. Since the size of the signature value affects the TLV-length of the packet, which also uses a variable size encoding, the cascading effect in sizes makes it tricky. Then, one has to choose between efficiency in time and memory: if we allocate a large enough buffer to handle the maximal possible size, there will be a memory waste; if we encode other parts first, compute the signature, and then reassemble them in the destination wire, we pay the cost of copying. However, the signature is always located at the end of a packet, which inspires a three-pass method using less memory:

- First pass is size estimation and buffer allocation. We assume the signature takes maximal possible space.
- Second pass is encoding. We encode the whole packet, compute the signature, and put them into the pre-allocated buffer.
- Third pass is fixing TLV lengths. We have two length numbers to fix: the length of the signature value and the whole packet. If the actual TLV-length needs a smaller than the maximal possible space, we simply move the type number of the packet down and truncate the head of the buffer.

The signing and verification of packets are out of the scope of this paper. Depending on specific implementations, there are multiple methods. For example, the decoder returns the indexing of wire regions covered by the signature for verification usage.

C EVALUATION

In this section, we evaluate the C++ template library by comparing it with other five implementations: three existing NDN libraries – ndn-cxx (in C++), python-ndn, YaNFD (in Go), and two referential implementations written by the authors — using Go reflection and code generation.[6] Among them, the encoding part of ndn-cxx and YaNFD is implemented by hand for all encodable types. Python-ndn and the two referential Go implementations also use automatic derivation, but do not follow the encoding model in this paper. We compare the six implementations in three aspects: performance, implementation complexity, and memory overhead. Performance is measured by the time consumed in encoding and decoding. To evaluate the complexity, we use lines of code (LOC), which is considered as one factor of user-friendliness and difficulty of maintenance. We show that our C++ template library based on automatic derivation has a comparative performance in all three aspects.

The future goal of our research is to have automatically derived code instead of taking the C++ template library, the numbers should be considered as a proof of practicality, instead of final performance results.

C.1 Experiment design

C.1.1 Methodology. We run each program against six test cases, and measure its execution time. The test cases consist of three encoding and three decoding. We use Data packets in all test cases; we offer a justification in §C.1.3. Two factors are related to the runtime: the size of the payload, and the length of the name. The former impacts the number of bytes flow through the program, and the latter the number of sub-elements. To identify these, we design the test cases as follows:

- The first is the control group, which has 100B of payload and a name of 3 components.

[6] These two implementations are written for the sake of evaluation. They are irrelevant with the model introduced in the paper, so the implementation details are omitted.

- The second has a larger payload (4kB) than, but the same name as the first one.
- The third has a longer name (33 components) than, but the same *total number of bytes* as the first one. To keep the total number of bytes flowing through the program the same, we decrease the payload.
- In encoding tests, the SHA256 hash is used for the signature field, because a Data packet is always signed and most signature types require computing a hash.

Each test case generates 10^6 packets. We run five times and take the average time. The standard error of the mean is less than 2% for all numbers.

C.1.2 Environment. We run all candidates on a Ubuntu 18.04 server, which has a Intel Core i7-5820K CPU (with 6 cores, but the programs we run are single-threaded) running at 3.3 GHz, and 64 GiB memory. We use gcc 10.3.0, go 1.6, and pypy 3.7 [7] to compile or execute.

C.1.3 Reasons for choosing data packets. A Data packet is simpler to parse than Interest, but contains everything that may count to performance: a name, a payload, and a nested data structure. Given that our model has limitations handling signatures (§ 4.2), we did not sign the Data packet. The goal of this evaluation is qualitatively showing the practicality of automatic derivation instead of investigating the performance of a specific implementation, so we believe unsigned Data packets are representative enough for this first evaluation.

C.2 Performance

The runtime data are shown in Table 1. Here, python-ndn is an outlier because Python is not optimized for performance.

To get more insights, we calculate the time increase rate over payload and the number of name components, shown in Table 2 by the columns named "/Payload" and "/Name", respectively. "Baseline" shows the estimated base of increase, i.e., the part of execution time that is irrelevant to payload size or name. Execution time on different machines can be very different, so the specific numbers do not make much sense. We mark outliers **bold** and only explain them column by column.

First, look at the two baseline time columns. Reflection solutions have higher numbers due to the cost of querying RTTI, about $3\mu s$. There is extra time consumed in python-ndn ($20\mu s$), probably because python-ndn uses a hash map to store temporary variables during encoding. That hash map uses strings as keys, which is time-consuming. About ndn-cxx and YaNFD, we did not observe a single bottleneck function or hot path. However, we found some parts that may contribute to it. For ndn-cxx, we noticed that during signing, the Data class parses the encoded wire back to make all pointers inside pointing to the new wire instead of their original value. This procedure takes about 20–25% of the total time, and should make up part of the baseline time. For YaNFD, we think it may come from the copying of sub-elements during encoding. In YaNFD's implementation of encoding, a TLV block first calls the encoding function of every sub-element, and then copies their wire into the allocated buffer. For example, Data contains Name as a sub-element, and Name contains NameComponent. Then, when Name's encoder is called, it first calls the encoding function of every NameComponent, and then copies the wire of NameComponent into the buffer of Name. In Data's encoding function, Name's wire is also copied. Thus, every NameComponent is copied twice. The deeper an element exists, the more times it needs to be copied.

Then, consider the two payload columns. Python-ndn is more sensitive to payload size when encoding. This may due to the use of the SHA256 algorithm. The Go and C++ implementations are directly compiled to machine code, but the Python version has a wrapper class in Python, which makes it slower. No one is sensitive to payload size on decoding, because the content is not copied.

Last comes the overhead accompanied with name components. In decoding, except for python-ndn [8], ndn-cxx and YaNFD give higher numbers. We think YaNFD's overhead is also due to its copying of sub-elements, as stated in the explanation of baseline time. As for ndn-cxx, we noticed that constructing a Name object from a string takes more than half of the execution time in test case 3. In the constructor of the Name class, making a shared pointer to a Buffer object costs 30% (i.e. 15% of the total time) Therefore, we guess it is the memory management that contributes to this overhead. In decoding, YaNFD is slightly worse than others. Our speculation is that, since every name component is treated as an object in garbage collector (GC), this leads to an increased workload of GC.

In summary, our C++ template library is competitive with other solutions in terms of execution time.

C.3 Implementation Complexity

Though LOC cannot fully represent implementation complexity, under equal readability conditions, fewer lines of code are generally less complicated and easier to maintain.

We consider the code of a TLV library as two parts: one contains basic data structures and helper functions that are used by every TLV object; the other is code written for a specific TLV object, such as Data. The second part is more critical, since this is the code a user needs to write when defining a new data structure.

Table 3 shows LOC data of all implementations. The "Library" column is the shared part. We inspected two TLV data structures, MetaInfo and Data (nested objects excluded). For each object, we further classify the code into three parts: the definition of data structure, the algorithms of encoding and decoding, and other code. MetaInfo is simple and straightforward. Data is complicated due to signature. These numbers are not precise, because the coding style may affect the LOC.

In the Library column, we can see that the three fully functional libraries — YaNFD, ndn-cxx, and python-ndn — has more lines of code than sample programs written by us. This is because semi-production code has more functionalities than test code, such as error handling and interfaces with other modules, so it is the expected result. In the algorithm column, ndn-cxx and YaNFD have more lines, as the user needs to manually call the encoding function of each field.

[7] We did not use CPython because it is known to be very slow. The time for each test case is 10 times longer than others.

[8] The algorithm that parses a string into a Name is implemented in Python, which is slower as expected.

Test Case		Encoding			Decoding		
		1	2	3	1	2	3
Number of Packets		1000000					
Total Encoded Size		157MB	4061MB	157MB	189MB	4093MB	189MB
Name Length		3	3	33	3	3	33
Golang	YaNFD	4.1s	**8.4**s	**15.8**s	2.5s	2.6s	8.4s
	Reflection	2.9s	3.9s	4.4s	3.3s	3.2s	5.0s
	Code Generator	0.4s	1.1s	1.8s	0.5s	0.6s	1.9s
C++	ndn-cxx	4.0s	4.5s	**22.1**s	1.1s	1.1s	2.7s
	C++ Template	0.6s	0.8s	3.5s	0.4s	0.5s	1.3s
Python 3	python-ndn	**12.9**s	**15.0**s	**32.3**s	4.0s	4.0s	4.6s

Table 1: Total runtime

	Encoding			Decoding		
	Baseline (μs)	/Payload ($\mu s/kB$)	/Name ($\mu s/1$)	Baseline (μs)	/Payload ($\mu s/kB$)	/Name ($\mu s/1$)
YaNFD	**2.76**	**1.10**	**0.39**	**1.86**	0.04	0.20
Reflection	**2.70**	0.25	0.05	**3.07**	0.00	0.06
Code Generator	0.23	0.18	0.05	0.33	0.02	0.05
ndn-cxx	**2.16**	0.13	**0.60**	0.96	0.00	0.05
C++ Template	0.30	0.05	0.10	0.29	0.02	0.03
python-ndn	**10.87**	**0.54**	**0.65**	**3.89**	0.02	0.02

Note: time numbers in this table are per packet values, so $1s$ in Table 1 becomes $1\mu s$.

Table 2: The baseline and increase rates of runtime for each Data packet

	Library	MetaInfo (def)	MetaInfo (alg)	MetaInfo (etc)	Data (def)	Data (alg)	Data (etc)
YaNFD	1052	6	**76**	**67**	10	**129**	**98**
Reflection	422	5	6	0	7	41	0
Code Generator	666	8	12	0	12	30	0
ndn-cxx	1890	7	**59**	**125**	9	**128**	**427**
C++ Template	653	9	6	0	13	32	0
python-ndn	909	4	0	12	16	53	11

Table 3: Numbers of lines of code breakdown

Test Case		1		2		3	
		Data	Wire	Data	Wire	Data	Wire
Encoded Wire Size		189		4093		189	
Name Length		3		3		33	
Golang	YaNFD [*Before encoding*]	575	-	4475	-	2255	-
	YaNFD [*After encoding*]	1719	213	13425	4117	4569	213
	Reflection & Code Generator	341	213	4241	4117	1061	213
C++	ndn-cxx [*Before Encoding*]	1655	-	5557	-	5735	-
	ndn-cxx [*After Encoding*]	12092	10372	12092	10372	14732	10372
	C++ Template	581	229	4481	4133	2501	229

Table 4: Memory overhead (bytes)

In future work, the shared library part should be automatically generated from our model, so the developers do not need to understand or maintain it. Taking that into consideration, the automatically derivation method can reduce developers' effort in the term of lines of code.

C.4 Memory Overhead

The runtime data are shown in Table 4 [9]. The "Data" column shows the total memory occupied by the data structure representing a Data packet, and the "Wire" column shows the memory used by the encoded wire. There is a canonical type `[]byte` to represent a binary string in Go, which is used in all three implementations to represent wire. Also, reflection and code generator have the same definition of data structures. Therefore, all Go implementations share the same numbers for "Wire" overhead, and reflection and code generator have also the same numbers for "Data". YaNFD and ndn-cxx combine original TLV blocks with encoded wires, so the sizes of "Data" are different before and after encoding.

Two C++ implementations have more overhead over Go due to memory management. In Go, a mark-and-sweep garbage collection is used, so every pointer to an object only contains a memory address (8B on a 64-bit platform). However, in C++, we use `shared_ptr` which maintains reference counting (24B on 64-bit). In ndn-cxx, there are many pointers maintaining the indices to sub-elements in the wire. As a result, every empty TLV block class uses 80B of memory. The example Data in Figure 1 and Figure 2 has 14 TLV blocks, so there is 1.1KB memory used by these TLV blocks, besides the data contained in this packet. Also, in ndn-cxx, every Data packet allocates a buffer of maximum size (8800B) for encoding due to implementation issues, so the memory usage after encoding is large. The user can copy the encoded wire into a buffer of exact length to reduce it.

This comparison shows that the memory overhead is basically orthogonal to the algorithm used in encoding or decoding. Instead, languages and memory management systems are main factors.

[9] Python object system is very complicated and known to have a very high memory overhead. It is also not easy to compute the total occupied memory of an object in Python. Therefore, we drop python-ndn in the table.