Transforming PEM-Encoded ASN.1 Cryptographic Objects into Human Readable JSON using Modular Arithmetic and Parse Tree

Muhammad Fatih Irkham Mauludi - 13524004
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: mhmd.fatih.im@gmail.com, 13524004@std.stei.itb.ac.id

***Abstract*— Modern cryptography has increased the security of the internet significantly. Popular cryptographic schemes such as symmetric/asymmetric encryption have become an integral part of internet communication protocols. In order to store or communicate cryptographic parameters between any involved parties, cryptographic protocol uses ASN.1 (*Abstract Syntax Notation one*) schema to define said parameters, then serialized it in DER (*Distinguished Encoding Rules*) or BER (*Basic Encoding Rules*), and finally encode it in Base 64 to become a PEM (*Privacy Enhanced Mail*) file. However, PEM content was not designed to be easily readable or modifiable without a different parser for each different ASN.1 structure. In this paper, the author would like to propose a method to parse PEM into JSON (*JavaScript Object Notation*) formatted file without any loss of structure information. This technique would allow us to modify or read elements within any PEM file by just using a regular JSON parser.**

Keywords*— Parsing; Tree; Rooted Tree, Abstract Syntax Notation One; Basic Encoding Rules; Distinguished Encoding Rules; Privacy Enchanced Mail; JavaScript Object Notation;

I. Introduction

Cryptography is an integral part of modern-day internet security. In fact, cryptography has been tracked to be around as far back as 1900 BC when it was used by the Egyptians to decorate the tomb of the dead. Modern cryptography first arises during the cold war period in the 1970s, and due to the invention of computers, the field of cryptography has received significant growth [1].

The world of cryptography revolves around the problem of hiding information using math principles. With the advancement of modern math and computational power, we saw the birth of modern crypto schemes, such as symmetric encryption, asymmetric encryption, key exchange protocol, secret sharing scheme, and digital signature generation.
However, with the growth of these complex protocols, a new problem has arisen. How can we represent cryptographic objects in such a way that we can reliably use them in communication protocol? That question was answered by the ITU (*International Telecommunication Union*) and IETF (Internet Engineering Task Forces), who introduced the standardization of ASN.1 schema and its encoding type (e.g. DER, BER, CER) to describe cryptographic parameters [2] [3] [4].

On the other hand, JSON is one of the most used markup languages used to store data. Its readability for both humans and computers makes it a good choice for a lot of developers to use. It is also more lightweight and faster than other markup languages such as XML (*Extensible Markup Language*) [5].

Every day, secure protocols such as HTTPS (*Hyper Text Transfer Protocol Secure*) and SSH (*Secure Shell*) are used by servers across the world. Such communication protocols highly utilize PEM as a way to send cryptographic data. Storing the PEM file as JSON formatted file might give an advantage in easier readability and modifiability in the long run, especially in web servers which heavily use concepts such as cookies and JWT (*JSON Web Token*).

Existing popular tools such as *OpenSSL* lack the ability to parse PEM directly into JSON, since it is mainly used directly to parse PEM directly into a running process memory.

In this paper, we will discuss a method to reliably parse a PEM file into a human readable JSON without any loss of information, using modular arithmetic and recursive descent parse tree. We will then implement it from scratch in the C programming language for efficiency. Lastly, we will discuss how this technology can be applied, and what we can improve from our current implementation method.

II. Theoritical Background

A. Modular Arithmetic

Modular arithmetic is a system of arithmetic in discrete math in which the value of the arithmetic group is limited to a certain field. This field is determined by a value called the *modulus*. A value x is an element of a modular arithmetic system with a modulus m, if and only if $0 \le x < m$. We can think of modular arithmetic as counting with “wrapping around”.

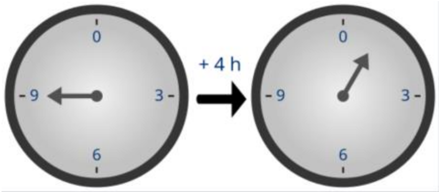


Fig 1. Example of a simple modular arithmetic on clock
Source:

Another important concept in modular arithmetic is *congruence*. We define that an integer a and b is called congruent modulo m if and only if:

$$m \mid (a - b) \mid (1)$$

We define that if an integer a and b is congruent modulo m, we will use the following notation:

$$a \equiv b\ (\mu o \delta\ m)(2)$$

Which we can read as "a is congruent to b in modulus m" or "a is congruent to b modulo m". Similarly, if two integer is not congruent, then we will use the following notation [6]:

$$a \not\equiv b\ (mod\ m)(3)$$

There are many applications of modular arithmetic in computer science, we will use the property of modular arithmetic in our implementation method later.

B. Graph, Tree, and Rooted Tree

A graph is a mathematical object used to represent the relationship between a discrete object and another discrete object. We define that a *vertex* is a discrete object, and an *edge* is a representation of the relationship between vertices. A graph is usually presented in a geometrical manner using a circle as a vertex and a line connecting the circle as an edge [7].

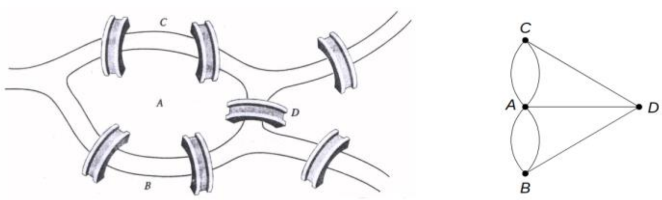


Fig 2. Classical Königsberg bridge graph reprsentation

Formally, a graph is defined as a tuple (V, E) where V is the set of all vertices, and E is the set of all edges. It must be satisfied that the set V must not be empty, but the set E can be empty [7].

We define a path of length n between two vertices v_1 and v_n to be a sequence of edges such that the first edge is connected to vertex v_1, the last edge is connected to v_n. We define a cycle to be a path which begins and ends at the same vertex, without repeating any of the edges in the sequence.

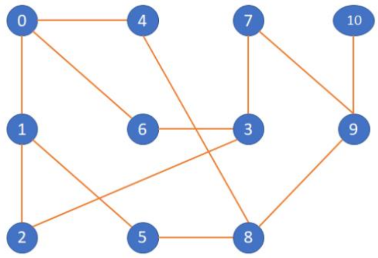


Fig 3. Example of a path and cycle on a graph

A tree is a graph which has no cycle, and there exists a unique path between any of the vertices. All trees are a subset of graph. Tree has a special property that if there are N vertices, then there will be exactly N-1 edges [8].

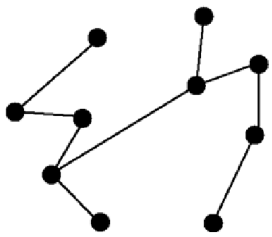


Fig 4. Example of a tree

A rooted tree is a tree in which a special vertex is chosen as a root and is assumed to have the first order of traversal. We can think of a root in a rooted tree as an "entry point" to enter the tree [9].

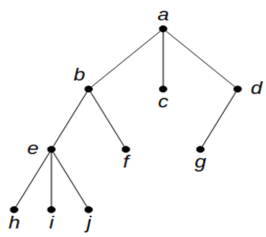


Fig 5. Example of a rooted tree

We will define the depth of a vertex in a rooted tree as the distance between the vertex and the root of the tree. We also define that if two vertices are connected, then the vertex with the higher depth is a child of the other vertex. Similarly, the vertex with the lower depth is a parent of the other vertex [9].

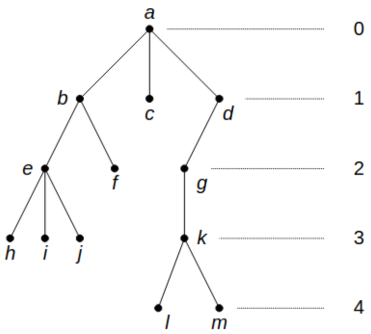


Fig 5. Example of a rooted tree and its depth properties

C. Abstract Syntax Notation One

Abstract syntax notation one is a formal notation used to define data and telecommunication protocol, which was standardized by the ITU. Formally, ASN.1 is a set of clauses which use the character from the ASN.1 character set [2].

A to Z	(LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z)
a to z	(LATIN SMALL LETTER A to LATIN SMALL LETTER Z)
0 to 9	(DIGIT ZERO to DIGIT 9)
!	(EXCLAMATION MARK)
"	(QUOTATION MARK)
&	(AMPERSAND)
'	(APOSTROPHE)
(	(LEFT PARENTHESIS)
)	(RIGHT PARENTHESIS)
*	(ASTERISK)
,	(COMMA)
-	(HYPHEN-MINUS)
.	(FULL STOP)
/	(SOLIDUS)
:	(COLON)
;	(SEMICOLON)
<	(LESS-THAN SIGN)
=	(EQUALS SIGN)
>	(GREATER-THAN SIGN)
@	(COMMERCIAL AT)
[	(LEFT SQUARE BRACKET)
]	(RIGHT SQUARE BRACKET)
^	(CIRCUMFLEX ACCENT)
_	(LOW LINE)
{	(LEFT CURLY BRACKET)
|	(VERTICAL LINE)
}	(RIGHT CURLY BRACKET)
‑	(NON-BREAKING HYPHEN)

Fig 6. Abstract Syntax Notation One character set

We can think of ASN.1 as a *pseudocode* to design data types which would be used in communication. Fundamentally, ASN.1 uses type-definition assignment clause in the form of,

<type> ::= <definition1> | <definition2> | <definition3> | ...

Which means that the type <type> can be defined as <definition1> or <definition2> or <definition3> and so on. A definition is just another type. Which means ASN.1 definition is naturally recursive. The base for this recursion is the ASN.1 built-in types [2]:

```
BuiltinType ::=
      BitStringType
    | BooleanType
    | CharacterStringType
    | ChoiceType
    | DateType
    | DateTimeType
    | DurationType
    | EmbeddedPDVType
    | EnumeratedType
    | ExternalType
    | InstanceOfType
    | IntegerType
    | IRIType
    | NullType
    | ObjectClassFieldType
    | ObjectIdentifierType
    | OctetStringType
    | RealType
    | RelativeIRIType
    | RelativeOIDType
    | SequenceType
    | SequenceOfType
    | SetType
    | SetOfType
    | PrefixedType
    | TimeType
    | TimeOfDayType
```

Fig 7. Abstract Syntax Notation One Built-in types

ASN.1 also has some keywords which gives special properties to defined type,

1. OPTIONAL, a type is optional in a definition
2. CHOICE, the definition is valid if at least one type is chosen
3. DEFAULT, defined a default value of a type in a definition
4. SEQUENCE, define a SequenceType
5. SET, define a SetType,
6. SIZE(MIN...MAX), the type is restricted to have size in the range of [MIN...MAX).

In cryptographic use, ASN.1 mostly uses the following built-in types:

1. BooleanType (True or False)
2. IntegerType (Arbitrarily big integer)
3. NullType (Type which indicate another type has no value)
4. OctetStringType (Sequence of byte character)
5. BitStringType (Sequence of bit character)
6. ObjectIdentifierType (String which uniquely identify another type)
7. DateTimeType (String in ISO time and date format)
8. SequenceType (Sequence of other type, order of element has a meaning)
9. SetType (Set of other type, order of element has no meaning)

Other than the built-in types, cryptographic objects also use the type ReferencedType such as:

1. UTF8String
2. IA65String
3. Printable String

4. T61String
5. Numeric String

These types can be used to define a cryptographic object as needed, for example here are the definitions of an RSA private key [10]:

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,  -- n
    publicExponent    INTEGER,  -- e
    privateExponent   INTEGER,  -- d
    prime1            INTEGER,  -- p
    prime2            INTEGER,  -- q
    exponent1         INTEGER,  -- d mod (p-1)
    exponent2         INTEGER,  -- d mod (q-1)
    coefficient       INTEGER,  -- (inverse of q) mod p
    otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

Fig 8.1. Example of ASN.1 structure in cryptography

```
OtherPrimeInfos ::= SEQUENCE SIZE(1..MAX) OF OtherPrimeInfo

OtherPrimeInfo ::= SEQUENCE {
    prime             INTEGER,  -- ri
    exponent          INTEGER,  -- di
    coefficient       INTEGER   -- ti
}
```

Fig 8.2. Example of ASN.1 structure in cryptography

D. Basic Encoding Rules

Basic encoding rules are the encoding rules standardized by ITU to serialize an ASN.1 structure into bytes which actually can be transferred in communication. The encoding rules follow a TLV (Tag-Length-Value) structure, in which a specific type will be encoded as a unique tag, followed by length of its content, and then the actual value of its content [3].

Table 1. Tag value for structure type in BER

Type	Tag Value
BooleanType	0x1
IntegerType	0x2
BitStringType	0x3
OctetStringType	0x4
NullType	0x5
ObjectIdentifierType	0x6
UTF8StringType	0xC
IA5StringType	0x16
UTCTimeType	0x17
SequenceType	0x30
SetType	0x31

A type can either be a primitive type or a constructed type. A primitive type cannot have another type inside its content, while a constructed type can (recursive).

The length component of a type encoding can have 3 forms [3]:

1. Short form:
   Length is described by 1 byte, must be indicated by the most significant bit having 0 value.

2. Long form:
   The first byte describes how many bytes will be used to describe the length. The number formed by bits 1-7 in the first byte, is the number of next consecutive bytes describing the length.

3. Indeterminate form:
   The first byte has its most significant bit set to 1, and all other bits set to 0. In this form, the content value will only end when 2 consecutive bytes with value 0 appear.

The content of a type also has special rules on how to construct them into serialized bytes, such as the following [3]:

1. BooleanType (primitive):
   The value must only be one byte in length. If the value is 0x00 then the Boolean value is False, if the value is any non-zero value, then the Boolean value is True.

2. IntegerType (primitive):
   The integer value can be formed by concatenating all content bytes and treating the first byte as the most significant byte, then turning the concatenated number into base 10.

3. BitStringType (primitive or constructed):
   The bit string value can be formed by concatenating all content bytes, then turning the concatenated number into base 2.

4. OctetStringType (primitive or constructed):
   The bit string value can be formed by concatenating all content bytes, then turning the concatenated number into base 16.

5. NullType (primitive):
   The content length must be 0 bytes. No value is to be processed. NullType only indicates the absence of value.

6. ObjectIdentifierType (primitive):
   Object identifier consists of some base 10 numbers separated by a '.' character. These number are called subidentifier. The first byte is formed by multiplying the first subidentifier by 40, then adding the result with the second subidentifier. All other subidentifiers are formed by concatenating consecutive bytes with MSB (*Most Significant Bits*) set to 1 with all bytes which has MSB set to 0 on its right.

All other string types are serialized normally by getting the unicode character represented by each byte on its content.

E. Distinguished Encoding Rules

Distinguished encoding rules are a subset of basic encoding rules with more restriction [3],

1. All type length must not be in the indeterminate form
2. A BooleanType with all bits set to 1 has the value True, if not then it has the value False.
3. Any string type must be primitive and not form a constructive type.

Most cryptographic objects use the DER encoding rules to serrialize ASN.1 to avoid ambiguity and to create a stricter parsing rule. However certain scheme such as the PKCS#8 (Public-Key Cryptography Standards number 8), uses the BER encoding [11].

F. Base 64 and Privacy Enhanced Mail

An ASN.1 Structure encoded in BER or DER will have a raw bytes value, which is not fit for communication protocol in some textual platform such as e-mail or text documents [12]. To reliably send binary encoded data using textual means, another encoding scheme must be used.

A Privacy Enhanced Mail has 3 components,

1. Header
2. Base64 Content
3. Footer

Each header is formatted as follows:

-----BEGIN <object>-----

Each footer is formatted as follows:

-----END <object>-----

Here, <object> can be any human readable string describing the base 64 encoded content. <object> between the header and footer must match. However, one thing to note is that in general, PEM file uses non-strict schema so that the <object> may not accurately the ASN.1 structure being encoded.

For example, <object> may describe a private key, but without any detail on what specific ASN.1 structure is used on the private key.

The base 64 encoding on the content is done by taking each consecutive 3 bytes value and turning it into 4 new bytes with six bits per bytes. The new bytes are then mapped into a list of predetermined values. If there is not enough 3 consecutive bytes, then a special padding character will be appended to the result.

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Fig 9. Base 64 character value set
Source: https://datatracker.ietf.org/doc/html/rfc4648

```
Input data:  0x14fb9c03d97e
Hex:      1    4    f    b    9    c     | 0    3    d    9    7    e
8-bit:    00010100 11111011 10011100  | 00000011 11011001 01111110
6-bit:    000101 001111 101110 011100 | 000000 111101 100101 111110
Decimal:  5      15     46     28        0      61     37     62
Output:   F      P      u      c         A      9      l      +
```

Fig 10. Base 64 Encoding Process Example
Source: https://datatracker.ietf.org/doc/html/rfc4648

III. Method

In this section, the author would like to discuss the implementation method which would be used to parse a PEM formatted file into JSON. All implementation details may not be able to be fully expressed in this paper due to large implementation complexity. Instead, the author would describe the general step in which implementation is taken.

The process will take the following steps:

A. Header and footer validation

In this step, user input will be validated to match the expected specification of a valid PEM file format. Unlike in general case, we will use the convention of a strict PEM header, so we can assume that the header/footer object will describe exactly what ASN.1 structure exists inside the content.

B. Base 64 decoding

Base 64 decoding is done by doing the reverse procedure of the previously described base 64 encoding. We will traverse each base 64 characters and determine the corresponding value through a lookup table, then each 4 bytes value (which has 6 bits per byte) will be concatenated to form 6 * 4 = 24 bytes. The 24 bytes will then be broken down into actual 3 bytes character of size 8 bits.

We can easily take the first 8 bits of the concatenated 3 bytes by using a modular arithmetic principle, which is to take the value modulo 256. Then divide it with 256. Keep repeating the procedure until the value is 0, then move into the next 4 consecutive 6-bit bytes.

```c
extern uint32_t base64_value[];

void base64_value_init(void);

uint32_t base64_decode(char *encoded, uint8_t *decoded);

uint8_t parse_pem(FILE *file, ParseTree *parseTree);
```

Fig 11. Function used in base64 decoding
Source: Author's archive

The following function is used in our implementation,

1. Base64_value_init: initialize the lookup table base64_value to be used inside base 64 decoding algorithm
2. Base64_decode: do base 64 decoding of PEM content
3. Parse_pem: validate PEM file, and then call base64_decode into the content of PEM file, then call the function to start building parse tree representation

C. Building Intermediate Parse Tree Representation

At this point, we will have an array of decoded base 64 values of the PEM content. The array should be a DER/BER encoded ASN.1 structure which we want to turn into a parse tree. Mathematically, we can see this process as a function which takes a sequence of numbers and turns it into a rooted tree structure.

We will build a parse tree based on a recursive descent parsing algorithm. This algorithm was chosen since ASN.1 is also a naturally recursive structure, so it makes sense to do recursive parsing.

```c
typedef struct Node {
    Tag tag;
    uint32_t length;
    uint8_t *content;
    uint32_t childNum;
    uint32_t childCapacity;
    struct Node **children;
} ParseTreeNode;

typedef struct {
    ParseTreeNode *root;
} ParseTree;
```

Fig 12. Parse tree structure
Source: Author's archive

```c
uint8_t init_parse_tree(ParseTree *parseTree);

void free_parse_tree(ParseTree *parseTree);

void visualize_parse_tree(ParseTree *parseTree, FILE *file);

ParseTreeNode *create_node(Tag tag, uint32_t length);

uint8_t append_children_node(ParseTreeNode *parent, ParseTreeNode *child);
```

Fig 13. Function used in parse tree generating
Source: Author's archive

The following function is used in our implementation,

1. Init_parse_tree: initialize memory of parse tree
2. Free_prase_tree: destroy memory of parse tree
3. Visualize_parse_tree: optional function which can be called if the user wants to see it
4. Create_node: create a node with a certain tag and length value
5. Append_children_node: attach a child node into a parent node

D. Building JSON file from Intermediate Parse Tree

Once the parse tree is built, we can start building a JSON file by doing a *preorder* traversal on the intermediate parse tree. By doing such traversal, we would be able to preserve the order of ASN.1 structure, and the JSON hierarchy would match the said structure.

Mathematically, we can think of this process as a function which takes a rooted tree of an ASN.1 structure and outputs another isomorphic rooted tree of a JSON structure.

```c
uint8_t parse_tag_type(Tag tag, char *parsedTag);

uint8_t parse_boolean(uint8_t *value, uint32_t length, char *parsedValue);

uint8_t parse_integer(uint8_t *value, uint32_t length, char *parsedValue);

uint8_t parse_null(uint8_t *value, uint32_t length, char *parsedValue);

uint8_t parse_object_id(uint8_t *value, uint32_t length, char *parsedValue);

uint8_t parse_bit_string(uint8_t *value, uint32_t length, char *parsedValue);

uint8_t parse_octet_string(uint8_t *value, uint32_t length, char *parsedValue);

uint8_t parse_general_string(uint8_t *value, uint32_t length, char *parsedValue);

uint8_t parse_utc_time(uint8_t *value, uint32_t length, char *parsedValue);
```

Fig 14. Function used in parsing PEM content
Source: Author's archive

Function to parse each expected ASN.1 built-in type will be prepared, and it will be called once a vertex has a matching tag with expected type. For example, if a vertex has a tag 0x2, the function parse_integer will be called to its content.

The last step is arguably the hardest one, to fully transform the PEM into JSON. We must find out what the right 'key' for each JSON value is. For example, in an RSA public key the first integer is a modulus 'm'. But in other ASN.1 structures, it is not.

Which means we must implement a tree-matching algorithm between our intermediate parse tree with many other expected ASN.1 structures. This is quite an impossible task for the author, so this implementation will stop here.

To ensure a unique key for each JSON key, a unique ID will be prepended to each key inside the generated file.

IV. Result and Discussion

The resulting parser has been tested using actual PEM files from open-source repositories, such as the Linux kernel repository (https://github.com/torvalds/linux) and OpenSSL source code repository (https://github.com/openssl/openssl).

A. OpenSSL DSA *(Data Signature Algorithm)* Private Key Example


```
-----BEGIN DSA PRIVATE KEY-----
MIIBugIBAAKBgQCnP26Fv0FqKX3wn0cZMJCaCR3aajMexT2GlrMV4FMuj+BZgn0Q
PnUxmUd6UvuF5NmmezibaIqEm4fGHrV+hktTW1nPcWUZiG7OZq5riDb77Cjcwtel
u+UsOSZL2ppwGJU3lRBWI/YV7boEXt45T/23Qx+1pGVvzYAR5HCVW1DNSQIVAPcH
Me36bAYD1YWKHKycZedQZmVvAoGATd9MA6aRivUZb1BGJZnlaG8w42nh5bNdmLso
hkj83pkEP1+IDJxzJA0gXbkqmj8YlifkYofBe3RiU/xhJ6h6kQmdtvFNnFQPWAbu
SXQHzlV+I84W9srcWmEBfslxtU323DQph2j2XiCTs9v15AlsQReVkusBtXOlan7Y
Mu0OArgCgYAapll6iqz9XrZFlk2GCVcB+KihxWnH7IuHvSLw9YUrJahcBHmbpvt4
94lF4gC5w3WPM+vXJofbusk4GoQEEsQNMDaah4m49uUqAylOVFJJJXuirVJ+o+0T
tOFDITEAl+YZZariXOD7tdOSOl9RLMPC6+daHKS9e68u3enxhqnDGQIUB78dhW77
J6zsFbSEHaQGUmfSeoM=
-----END DSA PRIVATE KEY-----
```

```
Tag: 0x30 (Sequence), Length: 442
│
├── Tag: 0x02 (Integer), Length: 1, content: 0x00
│
├── Tag: 0x02 (Integer), Length: 129, content: 0x00a73f6e85bf416a297df09f471930909a091dda6a3
9886ece66ae6b8836fbec28dcc2d7a5bbe52c39264bda9a7018953795105623f615edba045ede394ffdb7431fb5a
│
├── Tag: 0x02 (Integer), Length: 21, content: 0x00f70731edfa6c0603d5858a1cac9c65e75066656f
│
├── Tag: 0x02 (Integer), Length: 128, content: 0x4ddf4c03a6918af5196f50462599e5686f30e369e1e
40f5806ee497407ce557e23ce16f6cadc5a61017ec971b54df6dc34298768f65e2093b3dbf5e4096c41179592eb0
│
├── Tag: 0x02 (Integer), Length: 128, content: 0x1aa6597a8aacfd5eb645964d86095701f8a8a1c569c
6e52a03294e545249257ba2ad527ea3ed13b4e14321310097e61965aae25ce0fbb5d3923a5f512cc3c2ebe75a1ca
│
├── Tag: 0x02 (Integer), Length: 20, content: 0x07bf1d856efb27acec15b4841da4065267d27a83
│
├

{
  "Integer 0" : 0,
  "Integer 1" : 1174453852602609889728934756229521064198037940334763369370649067
63972925027543884810515009688752496683202339931512918752731591200270861283544758
80998445408992862458102329001280961862034384227808386290731386065495000164205728
06822732466275956287597978150195942755696990701960760410687748588182713000862344
02121,
  "Integer 2" : 1410281175091677582367253398738992577552560055663,
  "Integer 3" : 5468375615504379561362013392653733090072702027668098977711109145
67694502070162888513104618329702679160990107542743637210917674555882263819842209
71214628137801680107104413158235245699594853667977418115238639078798370136329344
42361947304971593089713884182157463975935464784835176306464627767636369364732490
6168,
  "Integer 4" : 1871412797039725202236732543553486983174878288378116126592593893
38762271633597079191883678685886109403018336070892515271887241399128725213303968
83010042149962502518380070851460188337690137391628575678002396949959347698456410
35363388382910140297296167821893384420749413934404105784192340609129825865263282
6649,
  "Integer 5" : 44224949390052197983749906902484910752360725123
}
```


B. OpenSSL ECC (Elliptic Curve Cryptography) Parameter Example


```
-----BEGIN EC PARAMETERS-----
MIIBHwIBATAlBgcqhkjOPQECMBoCAgFwBgkqhkjOPQECAwMwCQIBAQIBAgIBVTBg
BC7g0u4lCVIG9eKk+e0inx8lbnmg4rRVlw2NDYZb2Ud4xXbWLwq3UZzNKhqQauMN
BC78EhfUMgqQRSx2CljtzTDI3QabPDRFODejTtUMtUkX4cIRLYTRZPRE+PdHhgRq
BF0EEIXidVOB3MzjwVV6+hDC8MDCglZGxbNKOUy8+ovBayLn54npJ74hbwLh+xNq
X3s+sb3cumLV2LIFm1JXl/xzgixZBZxiOkX/OEPO6Ph80YVa2qgeKgdQuA/aIxAC
LQEAkFEtqa9ysINJ2Ypd1MewUy7KUc4D4tEPO3rFeb2H6QmuQKbxMenPzlvZZwID
AP9w
-----END EC PARAMETERS-----
```

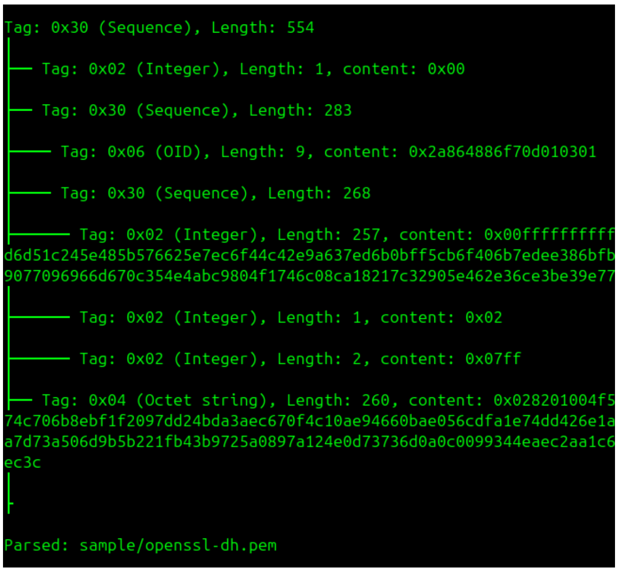
```
Tag: 0x30 (Sequence), Length: 287
│
├── Tag: 0x02 (Integer), Length: 1, content: 0x01
│
├── Tag: 0x30 (Sequence), Length: 37
│
├──── Tag: 0x06 (OID), Length: 7, content: 0x2a8648ce3d0102
│
├──── Tag: 0x30 (Sequence), Length: 26
│
├────── Tag: 0x02 (Integer), Length: 2, content: 0x0170
│
├────── Tag: 0x06 (OID), Length: 9, content: 0x2a8648ce3d01020303
│
├────── Tag: 0x30 (Sequence), Length: 9
│
├──────── Tag: 0x02 (Integer), Length: 1, content: 0x01
│
├──────── Tag: 0x02 (Integer), Length: 1, content: 0x02
│
├──────── Tag: 0x02 (Integer), Length: 1, content: 0x55
│
├── Tag: 0x30 (Sequence), Length: 96
│
├──── Tag: 0x04 (Octet string), Length: 46, content: 0xe0d2ee25095206f
│
├──── Tag: 0x04 (Octet string), Length: 46, content: 0xfc1217d4320a904
│
├── Tag: 0x04 (Octet string), Length: 93, content: 0x041085e2755381dcc
c623a45ff3843cee8f87cd1855adaa81e2a0750b80fda2310
│
├── Tag: 0x02 (Integer), Length: 45, content: 0x010090512da9af72b08349
│
├── Tag: 0x02 (Integer), Length: 3, content: 0x00ff70
│

{
  "Integer 0" : 1,
  "Sequence 1" : {
    "OID 2" : "1.2.840.10045.1.2",
    "Sequence 3" : {
      "Integer 4" : 368,
      "OID 5" : "1.2.840.10045.1.2.3.3",
      "Sequence 6" : {
        "Integer 7" : 1,
        "Integer 8" : 2,
        "Integer 9" : 85
      }
    }
  },
  "Sequence 10" : {
    "Octet string 11" : "0xe0d2ee25095206f5e2a4f9ed229f1f256e79
    "Octet string 12" : "0xfc1217d4320a90452c760a58edcd30c8dd06
  },
  "Octet string 13" : "0x041085e2755381dccce3c1557afa10c2f0c0c2
aa81e2a0750b80fda2310",
  "Integer 14" : 9194196556002283250851893699199753471497656173
  "Integer 15" : 65392
}

Parsed: sample/openssl-ecc.pem
```


C. OpenSSL DH (*Diffie Helmann*) Parameter Example


```
-----BEGIN PRIVATE KEY-----
MIICKgIBADCCARsGCSqGSIb3DQEDATCCAQwCggEBAP//////////yQ/aoiFowjTE
xmKLgNwc0SkCTgiKZ8x0Agu+pjsTmyJRSgh5jjQE3e+VGbPNOkMbMCsKbfJfFDdP
4TVtbVHCReSFtXZiXn7G9ExC6aY37WsL/1y29Aa37e44a/taiZ+lrp8kEXxLH+ZJ
KGZR7ORbPcIAfLihY78FmNpINhxV05ppFj+o/STPX4NlXSPco62WHGLzViCFUrue
1SkHcJaWbWcMNU5KvJgE8XRsCMoYIXwykF5GLjbOO+OedywYDoYDmyeDouwHoo+1
xV3wb0xSyd4ry/aVWBcYOZVJfOqVauUV0iYYmPoFEBVyjlqKrKpo//////////8C
AQICAgf/BIIBBAKCAQBPXxEkDA2EWknARF2EzUo6gc1eFNdKMVwa7aT3e2ClTIkN
B4Y6XsJCS5C4q0vKhHtdH5LswCxUPfTQQAOlKPzcdMcGuOvx8gl90kvaOuxnD0wQ
rpRmC64FbN+h503UJuGuNTFO2AvgLVb6EA637soAcWR6qLtRJ3wDpr1OW/ertIUj
jhzD1i255j+z6UVQBNLy882AUSHfjr1UzWTYfcyn1zpQbZtbIh+0O5cloIl6Ek4N
c3NtCgwAmTROrsKqHGmaW+pw4sOAAtNJByPT0y725s7tq4mAJKJgCc2J8Lbwbx9Z
s+tEoCidGYuBRNouVH6I6POwjIhdpU0kIscdv+w8
-----END PRIVATE KEY-----
```



D. Linux Private Key example

E. Discussion

All test results show that the implemented parser has been successfull in generating a JSON file with the same hierarchy as the PEM file, each value has been able to be parsed successfully, but each JSON key is not able to matched as we have discussed earlier. Overall the parser works as intended but still need improvement to become a full PEM to JSON converter. Each parse tree and generated JSON file share the same identical structure, which mean the structure is preserved successfully during the parsing

V. Conclusion

This paper explained the usage of modular arithmetic and rooted tree application inside parse tree to parse *privacy enhanced mail* formatted file into a more easily readable and modifiable JSON file. The implemented parser was able to preserve the ASN.1 structure successfully between transformation, when transforming PEM into parse tree and then transforming the parse tree into a JSON file.

However, the implemented parser was not able to correctly match each JSON key into the matching ASN.1 structure variable name, improvement could be made by implementing a tree matching algorithm which would try to match the received parse tree with a database of ASN.1 structure, and then try to predict what is the correct JSON key.

REFERENCES

[1] https://www.researchgate.net/publication/384113214_Historical_Evolution_of_Cryptography

[2] https://www.itu.int/rec/T-REC-X.680

[3] https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

[4] https://datatracker.ietf.org/doc/html/rfc8017

[5] https://www.researchgate.net/publication/379001324_Study_on_JSON_its_Uses_and_Applications_in_Engineering_Organizations

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/15-Teori-Bilangan-Bagian1-2024.pdf

[7] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdfh

[8] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf

[9] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf

[10] https://datatracker.ietf.org/doc/html/rfc8017

[11] https://datatracker.ietf.org/doc/html/rfc5208

[12] https://datatracker.ietf.org/doc/html/rfc7468

VI. APPENDIX

Here is the full parser implementation, which can be found on author’s GitHub: https://github.com/FieryBanana101/pem2json

ACKNOWLEDGMENT

Thanks to all.

[13]

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Juni 2025

Muhammad Fatih Irkham Mauludi - 13524004