マルチメディア通信と分散処理 111－10
コンピュータセキュリティ 20－10
（２００３．２．２７）

カスタマイズ可能な ASN.1 エンコーダジェネレータ

野田 敏達　小村 昌弘　金谷 延幸　中山 裕子

(株)富士通研究所 セキュアコンピューティング研究部
〒211-8588 川崎市中原区上小田中 4-1-1
E-mail: {binn, komura, kanaya, booko}@labs.fujitsu.com

あらまし　公開鍵証明書のデータフォーマットや多くのネットワークプロトコルの仕様は ASN.1 で記述され、データは符号化して取り扱われる。そのような符号化データを扱うアプリケーションでは、符号化データと、開発言語から直接操作可能なデータ構造の相互変換を行うプログラムルーチンが必要となる。我々は、アプリケーションに適切にカスタマイズしたプログラムルーチンを、仕様から自動生成することが可能なジェネレータの開発を行った。
キーワード　ASN.1 BER DER ジェネレータ

Customizable ASN.1 Encoder Generator

Bintatsu NODA,　Masahiro KOMURA,　Nobuyuki KANAYA,　Yuko NAKAYAMA

FUJITSU LABORATORIES Ltd., Secure Computing Lab.,
4-1-1, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan
E-mail: {binn, komura, kanaya, booko}@labs.fujitsu.com

Abstract　ASN.1 is commonly used to specify data formats for X.509 certificates and other protocols. To deal with data encoded in such formats, an application must have a program routine that converts the encoded data into a data structure suitable for a programming language. We developed a generator that can automatically generate such program routines from specifications, in a highly-customizable manner.
Keyword　ASN.1 BER DER generator

1. はじめに

Secure Sockets Layer (SSL) 通信 [12] の認証や電子署名で利用される公開鍵証明書 [5,6] のデータフォーマット、Simple Network Management Protocol (SNMP) [13] など多くのプロトコルの仕様は Abstract Syntax Notation One (ASN.1) [1,2] を用いて記述され、データは Basic Encoding Rules (BER) [3,4]、Distinguished Encoding Rules (DER) [4,5] 等に従い符号化して取り扱われる。このような符号化データを C 言語等の開発言語から直接操作することは符号化データの構造上困難であるため、一般的に、図 1 のように符号化データを開発言語上のデータ構造に変換して操作を行う。本稿では、符号化データから開発言語データ構造へ変換することをデコード、逆に開発言語データ構造から符号化データへ変換することをエンコード、それら変換を行うプログラムルーチンをエンコード/デコードエンジン (エンジン) と呼ぶ。

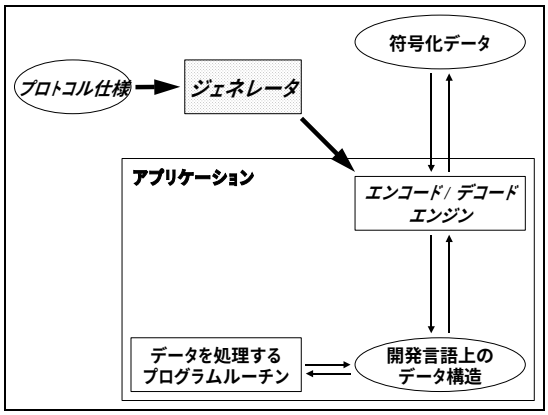


図 1. エンジンとジェネレータ

仕様からエンジンを手作業で実装するのは、単純ではあるが膨大な作業量が要求され、ミスも生じやすい。しかし、図 1 のように仕様からエンジンを自動生成するジェネレータを利用すれば、バグのないエンジンを容易に作成できる。

このようなジェネレータは既にいくつか存在する。しかし、従来のジェネレータが生成するエンジンは、生成段階でのカスタマイズができなかったため、開発するアプリケーション (例えば、リソースが限られている携帯端末用など) に最適なものにするには手作業での修正が必要だった。

そこで、我々はカスタマイズしたエンジンの生成が可能なジェネレータ、「プロトコルエンジンジェネレータ (PEG) 」を開発した。また、RFC2459 で定義されている仕様、PKIX1Explicit88 モジュール [6] に適用して、カスタマイズした 4 種類のエンジンを生成し、PEG のカスタマイズ機能の評価を行った。なお、PKIX1Explicit88 モジュールでは公開鍵証明書等のデータフォーマットが定義されている。本稿では、PEG のカスタマイズ機能および実装方式、評価結果について報告する。

2. 従来ジェネレータの概要と課題

2.1. 従来ジェネレータの概要

ASN.1 のモジュール (ASN.1 で記述された仕様) から C 言語で記述されたエンジンを自動生成するジェネレータには以下のものがある。

・(GNU) SNACC [7]
・eSNACC (Enhanced SNACC) [8]
・OSS ASN.1 Tools for C [9]
・OpenH323 [10]

従来ジェネレータの一つ eSNACC は、図 2 の ASN.1 で記述されている仕様から図 3 の C 言語の構造体、関数を生成する。

```
Certificate ::= SEQUENCE {
  tbsCertificate      TBSCertificate,
  signatureAlgorithm  AlgorithmIdentifier,
  signature           BIT STRING}
```

図 2. ASN.1 で記述された仕様

```
typedef  struct Certificate {
  struct TBSCertificate      *tbsCertificate;
  struct AlgorithmIdentifier *signatureAlgorithm;
  AsnBits                    signature;
} Certificate;
AsnLen DEncCertificateContent (...引数...)
{
    ...公開鍵証明書をデコードする処理...
}
void DDecCertificateContent (...引数...)
{
    ...公開鍵証明書をエンコードする処理....
}
```

図 3. eSNACC で生成されたエンジン

2.2. 課題

従来のジェネレータでは、生成するエンジンのカスタマイズはできないか、速度最適化、省メモリ最適化のせいぜい 2 パターンの選択ができる程度だった。そのため、開発するアプリケーションの機能やそのアプリケーションが使用される環境に適したエンジンを作成することはできず、以下の問題が生じていた。

問題 1. 不要な型に関するエンジンも生成

開発するアプリケーションでは不要であっても、仕様書で定義されている全ての型に関するエンジンを生成してしまうため、プログラムのサイズが大きくなってしまう。

問題 2. 生成する型が画一的

1. SNACC が生成するエンジンは、ASN.1 の INTEGER 型であるデータを C 言語の long 型に変換する。公開鍵証明書のシリアル番号は INTEGER 型と定義されているが、ベリサイン社の公開鍵証明書などのシリアル番号は 16 バイトであるため、SNACC で生成されたエンジンではこれらの公開鍵証明書をデコードすることができない。
2. eSNACC が生成する C++用エンジンは、INTEGER 型の 1 ワードを超えるデータを char[]型に格納することで、問題 2.1 を回避している。しかし、開発するアプリケーションによっては、INTEGER 型データが long 型で充分な場合もある。このような場合はオーバースペックとなり、速度、メモリ使用量のいずれの点においても好ましくない。

問題 3. 生成する処理が画一的

1. デコード処理中に何らかの処理を行いたい場合があるが、従来ジェネレータではそのような処理を組み込んだエンジンの生成はできなかった。行いたい処理には、例えば「データがアプリケーションの前提とする条件に当てはまらないことがわかり次第、ただちにデコード処理を打ち切る」などがある。
2. 符号化データの内部にデコードの必要のない部分が含まれる場合があるが、従来ジェネレータで生成されたエンジンは符号化データ全てのデコードを行ってしまう。

以上の問題から、従来は、生成されたエンジンのソースコードを手作業で修正する必要があった。しかし、それは複雑な作業で、バグの温床にもなりやすかった。特に、仕様が改版される場合や、アプリケーションがター

ゲットとするデータの変更を行う場合 (例えば、ある INTEGER 型の対象を 1 ワード以下から 1 ワード超へ変更を行う場合) には大きな修正作業が生じるなど、アプリケーションのメンテナンスを行うコストがとても大きかった。

3. プロトコルエンジンジェネレータ(PEG)

2.2 で挙げた課題を解決し、さらにユーザビリティを向上させるため、

1. 必要な ASN.1 の型の指定
2. 型実装のカスタマイズ
3. 処理実装のカスタマイズ

という機能を持った C 言語用エンジンのジェネレータであるプロトコルエンジンジェネレータ (PEG) を開発した。

3.1. カスタマイズ機能

以下にこれらの機能について説明する。

機能 1 必要な ASN.1 の型の指定 (← 問題 1)

従来ジェネレータは仕様で定義されている全ての型を対象としたエンジンを生成していたが、仕様内にはアプリケーションによっては不要な型の定義もあり、無駄なプログラムコードが生成されていた。

PEG ではアプリケーションで実際に使用する ASN.1 の型を指定することにより、

・指定された型
・指定された型が依存する型

だけを対象としたエンジンを生成する。

機能 2 型実装のカスタマイズ (← 問題 2)

従来ジェネレータでは、INTEGER 型は long 型というように、型に対応して生成する C 言語のデータ構造が決まっていたが、その型が最適ではない場合も多かった。

PEG では、任意の型およびその任意の要素に対して C 言語のデータ構造を変更できる。

型実装に関する代表的なカスタマイズには

・ポインタ参照、実体埋め込みの選択

がある。

機能 2.1 ポインタ参照、実体埋め込みの選択

構造型の要素をポインタ参照形式、実体埋め込み形式のいずれでもつのが適当であるかは、開発するアプリケーションによって異なるが、従来ジェネレータではポインタ参照形式のみであった。

PEG では、任意の型の任意の要素に対してポインタ参照形式、実体埋め込み形式を選択できる。

機能 3 処理実装のカスタマイズ (← 問題 3)

従来ジェネレータでは、型実装の場合と同様、型に対応して生成する処理ルーチンが決まっていたが、その処理が最適ではない場合も多かった。

PEG では、任意の型およびその任意の要素に対して処理ルーチンを変更できる。

処理実装に関する代表的なカスタマイズには

・処理の追加
・処理のスキップ

がある。

機能 3.1 処理の追加

従来ジェネレータが生成する処理ルーチンの途中で任意の処理を行うことができなかった。

PEG では、任意の位置に処理を追加記述できる。

機能 3.2 処理のスキップ

PEG では、任意の型およびその任意の要素に対応するデコード処理をスキップさせることができる。

3.2. 実現方式およびカスタマイズルール

図 4 のように、PEG はエンジン (C 言語のソースコード) を生成する Perl のコード (10,447 行) と、符号化データの解析処理などエンジンの基本処理ルーチンとなる C ライブラリのコード (3,941 行) からなる。また、仕様の解析に Perl のモジュール Parse-Yapp [11] を使用している。

PEG は、プロトコル仕様およびカスタマイズルールの入力を受け付け、Parse-Yapp を利用して仕様を解析し、Perl プログラムがカスタマイズルールを反映させてエンジンの生成を行う。生成されたエンジンは PEG の C ライブラリと連携して動作する。

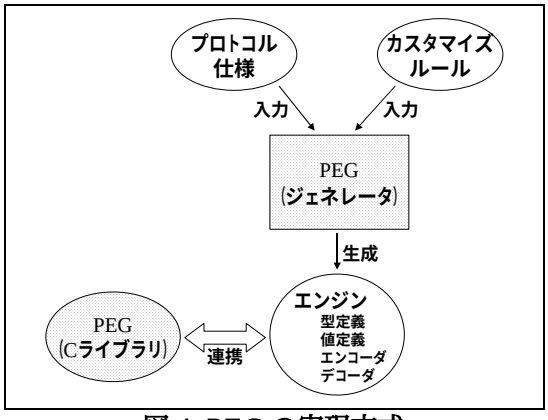


図 4. PEG の実現方式

3.1 で説明した各カスタマイズ機能を実現するためのカスタマイズは、図 5 のよう

な Perl で記述されたカスタマイズルールを用いて行う。このカスタマイズルールは 4 章で(d)スキップバージョンとして評価した。

4. 評価

PEG を使用して、RFC2459 で定義されている仕様、PKIX1Explicit88 モジュール [6] に対して、カスタマイズ内容を変えた 4 種類のエンジンを生成し、エンジンのカスタマイズが容易にできることの確認を行った。なお、このモジュールでは公開鍵証明書などのデータフォーマットが定義されている。また、生成されたエンジンの性能が、従来ジェネレータの一つである eSNACC で生成したエンジンと比較して劣っていない (むしろ優れている) ことの確認も行った。

4.1. カスタマイズ

PKIX1Explicit88 モジュールに対して、3.1 で説明した各カスタマイズ機能について以下のカスタマイズを行った。

1. 必要な ASN.1 の型の指定 (機能 1)

アプリケーションから直接扱う型は Certificate 型 (公開鍵証明書) のみとする。CertificateList 型 (CRL) は扱わない。

2. ポインタ参照、実体埋め込みの選択 (機能 2.1)

M (省メモリ用): 使用されるか不明な要素 (仕様で OPTIONAL と記述されている要素等) はポインタ参照、それ以外は実体埋め込みとする。

S (速度優先用): 全て実体埋め込みとする。

3. 処理の追加 (機能 3.1)

デコード時、公開鍵証明書のバージョンが v1 ならばデコード処理を打ち切る、という処理を追加する。拡張フィールドへの情報の記載を必須としているアプリケーションにおいては、拡張フィールドが存在しない公開鍵証明書のデコードを行う必要がない。このようにカスタマイズすることで、公開鍵証明書のバージョンが v1 (拡張フィールドが存在しないバージョン) であることがわかった時点で処理を打ち切れる。

4. 処理のスキップ (機能 3.2)

デコード時、以下の要素のデコードをスキップする。

Certificate.tbsCertificate.issuer
Certificate.tbsCertificate.subject
Certificate.tbsCertificate.extensions

issuer および subject は Name 型、extensions は Extensions 型である。Name 型はそれ以上デコードせずバイト列のまま扱ってもいい場合に、Extensions 型はその情報が必ずしも必要ではない場合に有効である。

以上のカスタマイズを組み合わせて表 1 のように 4 種類 (省メモリ、速度優先、打ち切り、スキップ) のバージョンのエンジンを生成した。なお、表の各機能に関するカスタマイズについて、○は行っていること、－は行っていないこと、M は省メモリ用のカスタマイズ、S は速度優先用のカスタマイズを行っていることを示している。

	カスタマイズ 1	2	3	4
(a)省メモリ	○	M	－	－
(b)速度優先	○	S	－	－
(c)打ち切り	○	S	○	－
(d)スキップ	○ (※)	S	－	○

※ Certificate 型のほかに Name 型、Extensions 型も指定している。

表 1 カスタマイズの組み合わせ

(d)スキップバージョンのカスタマイズルールを図 5 に示す。このカスタマイズルールでは Certificate 型、Name 型、Extensions 型およびそれらが依存する型を対象としたエンジンを生成する。また、生成されるエンジンの構造型の要素は全て実体埋め込みとし、Certificate 型の要素である tbsCertificate の要素の issuer、subject、extensions のデコードはスキップする。

```
package customize;
$Certificate = "PKIX1Explicit88.Ceritificate";
$tbsCertificate = "$Certificate.tbsCertificate";
# アプリケーションから直接扱う型の指定
$export = {
  "$Certificate" => TRUE,
  "PKIX1Explicit88.Name" => TRUE,
  "PKIX1Explicit88.Extensions" => TRUE
};
# ポインタ参照/実体埋め込みの選択
$pointer = {
  default => FALSE
};
#スキップする型、要素の指定
$skip = {
  "$tbsCertificate.issuer" => TRUE,
  "$tbsCertificate.subject" => TRUE,
  "$tbsCertificate.extensions" => TRUE
};
1
```

図 5. (d)スキップバージョンのカスタマイズルール

4 つのバージョンのカスタマイズルールの行数は表 2 のとおりである。このように数十行の簡単なカスタマイズルールを作成することで、アプリケーションに適切なエンジンを柔軟に生成できることを確認した。

(a)省メモリ	30 行
(b)速度優先	12 行
(c)打ち切り	29 行
(d)スキップ	20 行

表 2 カスタマイズルールの行数

4.2. 性能評価

PEG で生成した 4 つのバージョンのエンジンの生成コードサイズ、メモリ使用量、デコード速度、エンコード速度の性能測定を行い、カスタマイズが正しく機能していることの確認を行った。また、eSNACC で生成したエンジンとの比較も行い、性能が劣っていない(むしろ優れている) ことの確認も行った。評価に用いたプログラムは、生成されたエンジンを用いて公開鍵証明書 (符号化データ) をデコードし、さらにそのデコードしたものをエンコードする (入力された公開鍵証明書と同じものを出力する) というものである。

4.2.1. 測定環境および公開鍵証明書

測定環境を表 3 に示す。また、公開鍵証明書は DER で符号化された以下の 2 種類を用いた。公開鍵証明書の特徴を表 4 に示す。

1. Baltimore EZ by DST
2. GTE Cyber Trust Global Root

型	富士通 GP500S MODEL1000 CPU: UltraSPARC III (750MHz) メモリ: 512MB
OS	SunOS 5.8
C コンパイラ	gcc v2.8.1

表 3 測定環境

	Baltimore	CyberTrust
バージョン	v3	v1
シリアル番号のサイズ	4 バイト(*)	2 バイト(*)
公開鍵証明書のサイズ	1,030 バイト	606 バイト

(*)公開鍵証明書のシリアル番号は 4 バイト以下とした。これは、eSNACC が生成する C 用のエンジンは 4 バイトまでしか処理できないためである。

表 4 公開鍵証明書

4.2.2. 生成コードサイズ

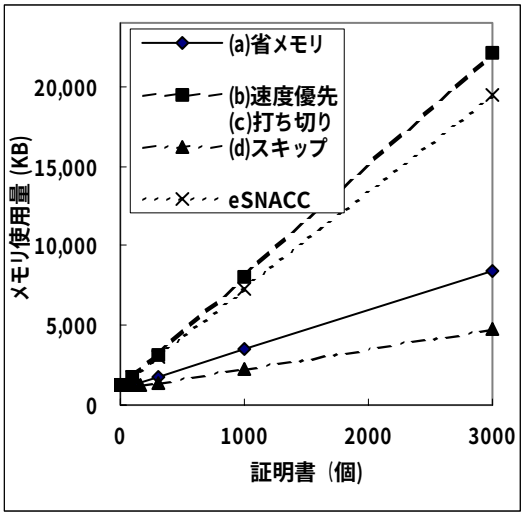
生成したエンジンのソースコードの行数と、このエンジンおよびライブラリを用いて作成した実行ファイルのサイズを測定した。なお、実行ファイルには strip を行っている。結果は表 5 のとおりである。PEG のいずれのバージョンも eSNACC の約半分となっている。カスタマイズ 1 により、PEG の方には CertificateList 型などが入っておらず、その効果が表れている。

		エンジン行数	ファイルサイズ
PEG	(a)省メモリ	1,713 行	48,972 バイト
	(b)速度優先	1,522 行	51,684 バイト
	(c)打ち切り	1,527 行	51,740 バイト
	(d)スキップ	1,445 行	50,188 バイト
eSNACC		10,639 行	116,296 バイト

表 5 エンジンのソースコードの行数と作成した実行ファイルのサイズ

4.2.3. メモリ使用量

公開鍵証明書 (Baltimore) の処理に必要なメモリサイズの測定を、どれだけ仮想メモリが使用できれば動作するかを測定することで行った。仮想メモリの制限は limit を用いて行った。結果はグラフ 1 のとおりである。



グラフ 1 メモリ使用量

公開鍵証明書のサイズが約 1 キロバイトと小さいこともあり、1 つのデータを処理するのに必要なメモリは eSNACC および PEG のいずれのバージョンも約 1,200 キロバイトでほぼ同じであった。しかし、同時に処理するデータが増えるに従い、(a)省メモリバージョンは 1 データにつき 2.5 キロバイト、(b)速度優先バージョンおよび(c)打ち切りバージョンは 7 キロバイト必要だった。また、eSNACC は 6 キロバイト必要だった。(b)速度優先バージョンがあまりよくないのは、必須ではないデータ用のメモリも確保しているからである。また、一部デコードを行わない(d)スキップバ

ージョンについては 1 データにつき 1.3 キロバイトの増加であった。

4.2.4. デコード速度

公開鍵証明書 (Baltimore および CyberTrust) のデコードに要する時間を測定した。測定はデータをメモリに読み込んだ後、デコードを 1 万回行い、その所要時間の平均から 1 回あたりのデコードに要した時間を算出した。その際、デコード時に動的に確保したメモリは毎回解放した。結果は表 6 のとおりである。

		Baltimore	CyberTrust
PEG	(a)省メモリ	59μs	43μs
	(b)速度優先	38μs	31μs
	(c)打ち切り	38μs	7μs
	(d)スキップ	15μs	13μs
eSNACC		119μs	95μs

表 6 デコード所要時間

(a)省メモリバージョン、(b)速度優先バージョン、eSNACC における速度の差は動的メモリ割り当ての回数によると考えられる。eSNACC では基本的に全てポインタ参照形式となっており、動的にメモリを割り当てる必要があるのに対し、(a)省メモリバージョンでは必須データは実体埋め込み、(b)速度優先バージョンは全て実体埋め込みとなっており、eSNACC と比較して動的メモリ割り当ての回数は格段に少なくできている。

(c)打ち切りバージョンにおける処理では、公開鍵証明書のバージョンが v1 である CyberTrust は処理が途中で打ち切られる。また、v3 である Baltimore は速度優先と同じ処理が行われる。処理が打ち切られる CyberTrust は、その結果が表れている。

(d)スキップバージョンは一部のデコードを行っていなため、高速である。処理後、必要な場合にだけスキップした部分のデコードを行うことで、全体としての処理速度が向上可能であると考えられる。

4.2.5. エンコード速度

C 言語のデータ構造 (Baltimore および CyberTrust の公開鍵証明書) のエンコードに要する時間を測定した。測定は公開鍵証明書をデコード後、デコードされたデータに対してエンコードを 1 万回行い、その所要時間の平均から 1 回あたりのデコードに要した時間を算出した。その際、エンコード時に動的に確保したメモリは毎回解放した。結果は表 7 のとおりで、eSNACC および PEG のいずれのバージョンもほぼ同じであった。

		Baltimore	CyberTrust
PEG	(a)省メモリ	71μs	63μs
	(b)速度優先	71μs	62μs
eSNACC		72μs	62μs

表 7 エンコード所要時間

動的メモリ割り当ての削減、処理の一部のスキップ、といった速度面に影響するカスタマイズは、デコードとは異なりエンコードには行っていないため、PEG と eSNACC の間に差はなく妥当な結果であると考えられる。

5. まとめと課題

本稿では、ASN.1 で記述された仕様からカスタマイズされたエンジンを自動的に生成するジェネレータの開発結果について報告した。

本ジェネレータは、数十行の簡単なカスタマイズルールを適用することで、開発するアプリケーションに適したエンジンを容易に生成可能である。また、生成したエンジンの性能もよい。

今後は、カスタマイズルールのより簡単な記述が可能になるように、改良を行っていきたい。

参考文献

[1] ITU-T Recommendation X.208: SPECIFICATION OF ABSTRACT SYNTAX NOTATION ONE (ASN.1)
[2] ITU-T Recommendation X.680: Abstract Syntax Notation One (ASN.1): Specification of basic notation
[3] ITU-T Recommendation X.209: SPECIFICATION OF BASIC ENCODING RULES FOR ABSTRACT SYNTAX NOTATION ONE (ASN.1)
[4] ITU-T Recommendation X.690: ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)
[5] ITU-T Recommendation X.509: THE DIRECTORY: AUTHENTICATION FRAMEWORK
[6] IETF Request for Comments 2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile
[7] SNACC: http://www.fokus.gmd.de/ovma/freeware/snacc/entry.html
[8] Enhanced SNACC: http://www.getronicsgov.com/hot/snacc_home.htm
[9] OSS ASN.1 Tools for C: http://www.oss.com/products/tools.html
[10] OpenH323: http://www.openh323.org/
[11] Parse-Yapp: http://search.cpan.org/author/FDESAR/Parse-Yapp/
[12] IETF INTERNET DRAFT: The SSL Protocol Version 3.0
[13] IETF Request for Comments 1157: A Simple Network Management Protocol (SNMP)