逆方向実行可能言語によるエンコーダとデコーダの同時実装

○田中 哲[1,a)]

概要：本発表ではプログラムを逆方向にも実行可能なプログラミング言語 Conservation を提案し、例として X.509 証明書のエンコーダとデコーダを記述する。また応用として、エンコーダを修正して間違った証明書を生成し、既存の暗号ライブラリに対してファジングを行う。通信プロトコルやデータフォーマットなど、データをプロセスの外部で表現するためにはデータをバイト列にエンコードし、また、逆にバイト列をデータにデコードすることが欠かせない。エンコードとデコードは対になる処理であるが、通常は別々に開発するため、正しく対応がとれた実装になるとは限らない。本発表ではプログラムを逆方向にも実行可能なプログラミング言語によりエンコードとデコードをひとつのプログラムとして記述可能とする。それにより常に正しく対応のとれたエンコーダとデコーダが開発できる。また、開発における利点だけでなく、ファジングにも利用できることを示す。

Implementing Encoder and Decoder at Once in a Reversible Programming Language

Akira Tanaka[1,a)]

Abstract: This presentation proposes a reversible programming language: Conservation. We implemented encoder and decoder for X.509 certificates as an example program. Also, we modify the encoder to generate wrong certificates and fuzz existing cryptographic library. Implementation for communication protocols and data formats needs encoder (converter from a data structure to a sequence of bytes) and decoder (converter from a sequence of bytes to a data structure). Usually they are implemented individually and not guaranteed to be inverse functions of each other. This presentation explains the language can be used to implement a pair of encoder and decoder with single program. The program can run forward to work as an encoder and run backward as a decoder. The behaviors are guaranteed to be inverse functions. This means correct program can be implemented with less effort. The program can also be used for fuzzing by modifying the encoder.

1. はじめに

本論文では逆方向実行可能プログラミング言語 Conservation を提案する。

通信プロトコルやデータフォーマットなど、プログラムがデータを外部と入出力する場合には、データ構造とバイト列を変換しなければならない。入出力にはエンコーダ (データ構造からバイト列への変換関数) とデコーダ (バイト列からデータ構造への変換関数) の両方が必要で、それらは対になる動作をする。

一般に、エンコーダとデコーダは別々に実装する必要がある。しかしこれはひとつの仕様からふたつのプログラムを作っているという意味で本質的には冗長である。

そこで、ひとつの記述がエンコーダとデコーダの両方として動作できれば、この冗長性を取り除くことができる。そうすれば、より短い記述で同等な機能を実現できる。これはバグが混入する可能性を減らし、記述の高品質化につながる。

そのような記述は BNF のような構文定義では充分でな

[1] 独立行政法人 産業技術総合研究所 セキュアシステム研究部門
Research Institute for Secure Systems, National Institute of Advanced Industrial Science and Technology (AIST)
[a)] tanaka-akira@aist.go.jp

い。これは実際の通信プロトコルやデータフォーマットには BNF では十分に記述できない要素があるためである。いくつか例を示すと以下のようなものがある。

- 整数で長さが指定され、その後に指定された長さだけのデータが続く形式
- 1 バイトの中にビット単位で複数の情報が詰め込まれている形式
- 構造を持ったデータが圧縮や暗号化されて格納されている形式

このような形式は BNF だけでは表現できず、扱うにはより一般的な言語が必要になる。

そこで、逆方向にプログラムを実行できるプログラミング言語を考える。そのようなプログラミング言語があれば、入力と出力を入れ替えて実行でき、ひとつのプログラムがエンコーダとデコーダの両方として利用できる。

逆方向に実行できるということは、通常のプログラミング言語と異なり、処理の途中で情報を失うことができないということを意味する。情報を失うと出力から入力を再現できなくなるからである。たとえば、代入文で変数の値を破壊的に書き換えると、実行後の状態から実行前の値を得ることができず、逆方向の実行ができなくなる。つまり、少なくとも単純な代入文は使えないことになる。

そのため、以下の性質を持つプログラミング言語 Conservation を提案する。

- エンコーダとデコーダの両方をひとつのプログラムで実現するために、決定的に逆方向に実行できる
- 逆方向に実行可能であるために情報を失う代入文がなく、副作用を持たない
- 長さやビットの操作のために、整数演算を行える
- 再帰下降構文解析のために、再帰とローカル変数が扱える
- バイト列と構文木を表現するデータ構造が扱える

また、データ構造とバイト列の対応の定義は、構文の仕様とみなせるので、エンコーダとデコーダ以外の利用法も考えられる。本論文ではファジングに利用できることを示す。

2. 自然数のエンコーダ・デコーダ

まず自然数と文字列を相互に変換するエンコーダ・デコーダを例として Conservation を説明する。簡単のため、ここで自然数は 1 以上の整数とし、また文字列のかわりに一桁の整数のリストを扱う。

Conservation では値として整数、リスト (空リストと cons セル) を扱える。自然数のエンコーダは整数を一桁の整数のリストに変換するものとし、デコーダは一桁の整数のリストを整数に変換するものとする。

エンコーダは整数を 10 で割って商と余りを求め、余りを最下位桁とし、商を再帰的に処理し、上位桁のリストと最下位桁を cons() でつなげることで整数のリストを作る。この再帰は整数が 0 になるまで行われる。リストの i 番目の要素が 10^i の桁の数字となる。

デコーダは逆に整数のリストをひとつの整数に変換する。これも再帰的に処理され、リストが空リストになるまで行われる。

自然数のエンコーダ・デコーダは図 1 のように定義できる。`nat2str` 関数を順方向に動かしたときにはエンコーダとして動作し、逆方向に動かした時にはデコーダとして動作する。順方向に動かした場合、順方向引数 `n` から逆方向引数 `s` に変換する。逆方向に動かした場合は反対に `s` から `n` に変換する。関数名直後の括弧内には単方向引数を記述できるが、`nat2str` 関数では空である。

図 1 では、順方向に動かしたときの動作をコメントとして付記してある。順方向に動かしたときは、基本的には上から下に制御が流れるが、逆方向に動かしたときは逆に下から上に制御が流れる。逆方向の場合の動作を図 2 にコメントで示す。

関数は失敗することができ、この失敗は条件分岐に利用される。また、条件分岐で選べる選択肢が存在しなければ失敗が発生する。図 1 の分岐 (`cond`) では `bind`, `guard`, `cons` というプリミティブ関数の失敗を利用している。`nat2str` 自身は順方向に動かしたときに `n` が負であれば失敗するが、この失敗は利用していない。

`bind` は順方向では変数の定義として働き常に成功する。しかし逆方向では変数の値を検査する表明となって失敗しうるので条件判定に利用できる。図 1 の最初の `n -> !bind(0)` という呼び出しには関数名の前に `!` がついているが、これは呼び出し元とは反対方向に動作させる指定である。その結果、順方向に `nat2str` が動いているときには `bind` は逆方向に動くことになり、条件判定として利用できる。具体的には `n` が 0 でなければ失敗する。

`nat2str` の逆方向引数 `s` には最上位桁が 0 でないという制約がある。たとえば、402 という整数をエンコードすると `[2, 0, 4]` というリストになる。このとき、`[2, 0, 4, 0]` など、上位桁に余計な 0 がつくことはない。逆に、上位桁に余計な 0 がついたリストをデコードすると失敗する。実際のタイミングとしては再帰が進んで `[0]` をデコードする時に失敗する。`[0]` は空リストではなく cons セルなので、条件分岐では右ガード `{ r, t -> cons() -> s; }` が成功して選ばれる。しかし、実行が進むと対応する左ガード `{ guard(0 < n); }` が失敗するので入力が正しくなかったことが分かる。条件分岐が終わったときに反対方向に戻れるか確認することで、不適切な入力を拒否できる。これにより成功して得られた結果は確実に入力に戻せることが保証できる。

このように、Conservation では、リストというデータ構造を扱い、決定的に逆方向に動作できるプログラムを記述

```
define n -> nat2str() -> s      # nat2str の定義開始。順方向に呼び出されるなら n を受け取って s を返す
  cond                          # 分岐
  when # n が 0 のときの処理
    { n -> !bind(0); }          # n が 0 なら成功して続きを行う。そうでなければ失敗
    {}                          # なにもしない
    { bind([]) -> s; }          # s に空リストを束縛
  when # n が 0 よりも大きいときの処理
    { guard(0 < n); }           # n が 0 よりも大きければ成功して続きを行う。そうでなければ失敗
    {
      n -> divmod(10) -> q, r;  # n を 10 で割り、商を q、余りを r に束縛
      q -> nat2str() -> t;      # nat2str() を再帰的に呼び出し、q を変換した結果を t に束縛
    }
    { r, t -> cons() -> s; }    # r と t を使って cons セルを生成し、s に束縛
  end
end
```

図 1 nat2str の定義と順方向の動作

```
define n -> nat2str() -> s      # 逆方向に呼び出されるなら s を受け取って n を返す
  cond                          # 分岐
  when # s が空リストのときの処理
    { n -> !bind(0); }          # n に 0 を束縛
    {}                          # なにもしない
    { bind([]) -> s; }          # s が空リストなら成功して続きを行う。空リストでなければ失敗
  when # s が cons セルのときの処理
    { guard(0 < n); }           # n が 0 よりも大きいことを確認（表明）
    {
      n -> divmod(10) -> q, r;  # q * 10 + r を計算して n に束縛
      q -> nat2str() -> t;      # nat2str() を再帰的に（呼び出し元と同じく逆方向に）呼び出し、t を変換した結果を q に束縛
    }
    { r, t -> cons() -> s; }    # cons セル s を分解して r と t に束縛。s が cons セルでなければ失敗
  end
end
```

図 2 nat2str の定義と逆方向の動作

できる。

3. 言語の詳細

Conservation の各構成要素について述べる。

なお、Conservation で提供している変数は関数単位のローカル変数だけである。変数は関数呼び出しのたびに用意される変数環境に記録される。

3.1 関数定義

両方向関数の定義は以下の形で行われる。

`define` 順方向仮引数の並び `->` 関数名（単方向仮引数の並び）`->` 逆方向仮引数の並び

文

`end`

変数の並びと矢印 (->) は、並びが空なら省略可能である。

両方向関数が呼び出されたときの動作は以下のとおりである。

(1) 呼び出し元の変数環境においていくつかの変数を「消費」する。
(2) 単方向引数として与えられた式をそれぞれ評価する。
(3) 消費した変数の値と単方向引数の値および対応する仮引数から関数内部の変数環境を作る
(4) その変数環境を元に関数本体の文がなんらかの計算をする。
(5) 変数環境から計算結果を取り出し、呼び出し元の変数環境にいくつかの変数を束縛する。

ここで変数の消費というのは変数を参照して値を取り出すと同時に変数束縛を解いてそれ以降使用できなくするという意味である。

順方向に動作する場合、呼び出し元で順方向仮引数に対応する実引数変数が消費され、逆方向仮引数に対応する実引数変数が束縛される。逆方向に動作する場合には逆方向引数が消費、順方向引数が束縛となる。

また、関数は失敗することができる。これは条件判断に用いられる。

関数が成功する場合、最後に変数環境に残っているすべての変数の値を呼び出し元の変数に束縛しなければならない。

たとえば、`nat2str` は `n -> nat2str() -> s` と定義されているので、順方向の呼び出し時にはひとつの変数を消費し、値が `n` に束縛される。括弧内の仮引数は存在しないため、`n` のみを元に関数本体によって計算が行われ、計算が終了した時点の変数 `s` の値が呼び出し元の変数に束縛される。ここで、`s` 以外のすべての変数は消費されるため、残っているすべての変数を呼び出し元に戻すという制約を満たす。また、逆方向の呼び出し時には消費された変数の値が `s` に束縛され、計算結果として `n` に得られた値が呼び出し元で束縛される。

3.2 順次実行

以下のように複数の文をセミコロンで区切ってひとつの文にまとめることができる。

文; 文; ... 文;

空列もひとつの文とみなす。

順方向実行なら左から、逆方向実行なら右から実行される。

3.3 条件分岐

条件分岐は以下の形式の cond 文で行う。

```
cond
when { 文 } { 文 } { 文 }
...
end
```

条件分岐の各選択肢は左ガード、分岐本体、右ガードからなる節である。

なお、左ガードおよび右ガードは `{ 文 }` のかわりに `otherwise` と記述しても良い。ただし、`otherwise` はひとつの条件分岐につき、左右それぞれでたかだかひとつの選択肢にしか使用できない。`otherwise` は他のすべての選択肢のガードが失敗したときのみ成功するガードとみなす。

順方向実行では、分岐開始時の変数環境を使ってすべての選択肢の左ガードが実行される。ここで選択肢のひとつのみが成功すると、その選択肢が選ばれる。その左ガードを実行した結果の変数環境を使って分岐本体が実行され、さらに右ガードが実行される。ガード内では関数を呼び出して変数環境が変化する可能性があるので、右ガードも実行しなければならない。

なお、右ガードが実行された後の変数環境を使って、逆方向に各選択肢の右ガードを実行した場合、選択された右ガードのみが成功し、他の右ガードはすべて失敗しなければならない。

逆方向実行では反対に右ガードで選択肢が選ばれ、選ばれた選択肢の分岐本体と左ガードが実行される。

たとえば、`nat2str` の順方向実行では、`n` の値によって分岐する。`{ n -> !bind(0); }` という左ガードで `bind` 関数を使って 0 であることを判定し、`{ guard(0 < n); }` という左ガードで `guard` 関数を使って 0 よりも大きいことを判定する。0 よりも小さかったら失敗である。なお、前者の `n -> !bind(0)` は変数 `n` の束縛を解く。

3.4 関数呼び出し

関数呼び出しは以下の 2 つの形式のいずれか記述する。

- 順方向引数変数の並び `->` 関数名（単方向引数式の並び）`->` 逆方向引数変数の並び
- 逆方向引数変数の並び `->` `!`関数名（単方向引数式の並び）`->` 順方向引数変数の並び

変数の並びと矢印は、並びが空なら省略可能である。

両方向関数は順方向でも逆方向でも呼び出せるので、その区別を関数名直前の ! の有無で行う。! が無ければ、呼び出し元と同じ方向で呼び出し、有れば反対方向で呼び出す。つまり、呼び出し元が順方向なら、! 無しなら順方向、有りなら逆方向で呼び出される。呼び出し元が逆方向なら、! 無しなら逆方向、有りなら順方向で呼び出される。

呼び出される関数が順方向に動作するなら、順方向引数変数を消費し、結果を逆方向引数変数に束縛する。逆方向に動作するなら、逆方向引数変数を消費し、結果を順方向引数変数に束縛する。いずれにしても、呼び出し元が順方向で動作していれば、左側の変数を消費し、右側の変数を束縛する。

たとえば、nat2str 内の、n -> !bind(0) は bind 関数の反対方向の呼び出しである。つまり nat2str が順方向に動作していれば、bind は逆方向に動作する。bind 関数は順方向に動かすと単方向引数をそのまま逆方向引数変数に束縛する。これを逆方向に動かすと、逆方向引数変数 n を消費し、単方向引数と等しい場合に成功し、異なれば失敗する。nat2str ではこれを n が 0 かどうかで分岐する判定に利用している。

3.5 単方向式

関数呼び出しの括弧内には単方向引数の実引数として単方向式を記述できる。ここでは整数、文字列、リストなどのリテラル、変数参照、通常の (単方向の) 関数呼び出し、通常の 2 項演算子などを利用できる。単方向式は逆方向には動作せず、通常の評価だけで副作用もないので、本論文では詳細について省略する。とくに、単方向式内部の変数参照は束縛を解かない (変数を消費しない)。

なお、ある関数呼び出しで消費する変数はその関数呼び出しの単方向式では利用できない。これは、単方向式はどちらの方向でも同じ動作をするため、参照する変数は事前に存在していなければなないのに対し、消費される変数はその反対方向に動作したときは関数呼び出しの最後に束縛される変数であり、事前には存在しないからである。

4. X.509 証明書のエンコーダ・デコーダ

X.509 証明書 [6][5] は TLS[4] という暗号化通信などで用いられる証明書である。暗号化通信で証明書が通信相手から送信された場合、証明書を解析して署名などを取り出す必要がある。つまり通信相手から送られたデータを扱うため、証明書の解析コードにバッファオーバーフローなどの問題があると脆弱性となりうる。そのため、そのようなライブラリは可能な限り網羅的にテストすることが望ましい。

ここで「網羅的」というのは証明書の仕様を網羅するという意味であれば、実装が仕様に合致しているかテストしているといえる。(これに対し、テストカバレッジの網羅性は実装に対する網羅性であって、仕様に対する網羅性ではない。) ところで、本論文で延べた言語で証明書のエンコーダ・デコーダを記述すれば、それは証明書の構文の仕様とみなせる。そこで、これを利用するテストを考えた。

本節ではこのために記述したエンコーダ・デコーダについて述べる。

X.509 証明書は ASN.1[7] で記述された構文により定義される。ASN.1 は文法を記述する言語であり、構文木を具体的なバイト列に変換する方法は複数存在する。X.509 証明書の場合は DER (Distinguished Encoding Rules)[8] という方法で構文木をバイト列に変換する。DER は BER (Basic Encoding Rules)[8] のサブセットであり、BER に対して同じ構文木は同じバイト列に変換されるように制約を加えたものである。

そこで、DER のエンコーダ・デコーダを実装した。実際のコードを付録に示す。asn1_parse_objs は順方向でバイト列から構文木に変換する関数である。

バイト列は整数のリストであり、構文木は mknode 関数で生成するデータ構造である。

5. ファジング

ファジング [3] とは正しくないデータをプログラムに与えて、適切にエラー処理を行えるか確認するテスト手法である。ここでは認証局証明書で署名したサーバ証明書をデコードして構文木を生成し、その構文木をエンコードする際に細工をして正しくないサーバ証明書を生成した。その正しくないサーバ証明書を以下の 2 つの方法で OpenSSL[9] ライブラリに使用させた。

- 認証局証明書でサーバ証明書の署名を検証する。
- サーバ証明書を使って TLS による通信路を確立する。

前者は純粋に証明書を扱う機能に関するテストである。後者はより実際的な利用法に近い形態のテストである。どちらも正しくない証明書を使っているので、検証失敗、通信路確立失敗が期待される動作である。期待されない動作は、検証成功、通信路確立成功、そしてテストプログラムの異常終了である。

一般にファジングと呼ばれる技法は乱数を用いてデータを生成するなど、必ずしも間違ったデータを生成するとは限らないやりかたも含む。その場合、正しいデータが生成されてしまうこともありうるため、異常終了しないという以外、なにが正しい動作かは自明でない。しかし、本論文で記述したファジングはすべて、なんらかの意味で間違った証明書を確実に生成する。そのため、検証失敗、通信路確立失敗という動作を確実に期待でき、そうならなければライブラリに問題があることがわかる。

5.1 ファジングのための言語拡張

X.509 証明書に対するファジングのため、言語に以下の文を追加した。

backward_fuzzing(名前) { 変数の値を変えるコード }
backward_fuzzing(名前) { 変数の値を変えるコード } if ファジングを適用する条件

これらは逆方向実行時に、変数の値を変更することによって本来の結果とは異なる結果を作り出す。括弧内の名前は適用したファジングを集計するなどの目的に使うもので、動作には影響しない。ファジングを適用する条件が記述されていれば、その条件が成り立ったときにファジングの適用が許可される。条件が記述されていなければ、常に許可されるとみなされる。適用される時には変数の値を変えるコードにより変数の値が変化する。

変数の値を変えるコードには tokens = []; などの代入を記述する。この例では tokens 変数に空リストを代入している。

asn1_parse_objs の逆方向実行は構文木からバイト列への変換なので、その変換中に backward_fuzzing が適用されれば本来とは異なるバイト列を作り出すことになる。

なお、backward_fuzzing は順方向実行には影響を与えない。

一回の変換では複数回 backward_fuzzing が実行され、それらがファジングの適用の候補となる。その中で実際にどれを適用するかは外部からの設定で制御する。まったく適用しないことも、すべて適用することもできる。

ファジングが適用された場合、変数の値が変化するが、これが原因で意図しない失敗が発生して結果が得られないことがある。これを防ぐため、ファジングを行った後、以下の 2 箇所で失敗を抑制する。

- guard 関数はスキップし常に成功する。ただし条件分岐の左ガード・右ガード内での呼び出しは除く。
- 条件分岐終了時に反対方向に戻れることの確認はしない。

なお、backward_fuzzing と対になるものとして順方向実行時にファジングを行う forward_fuzzing を定義することも容易であるが、本論文では利用しないので省略する。

5.2 定義したファジング

X.509 証明書のエンコーダ内に以下の 8 種類のファジングを定義した。

- premature_end_at_content
- premature_end_at_content_length_beginning_octet
- premature_end_at_hi_tag_number_octet
- premature_end_at_long_content_length
- too_long_integer_encoding
- too_long_length_encoding
- too_long_length_encoding_max
- too_long_tagnum_encoding

この中で premature_end_at_content_length_beginning_octet と too_long_tagnum_encoding を以下に説明する。

premature_end_at_content_length_beginning_octet は途中で途切れている証明書を生成する。証明書はバイトのリストで表現されるので、後ろから構成されていく。そこで、途中まで構成したリストを空リストに置換すれば途切れた証明書が構成できる。それを行っている例が asn1_parse_content_length 関数内にある図 3 のコードである。ここで、tokens = []; によって途中まで構成したリストを空リストに置換している。

証明書の中には整数が埋め込まれるが、整数は文脈によって決まる形式で表現しなければならない。ただし、それらの形式は BER では冗長な表現が可能であり、DER では冗長な表現が禁止される。たとえば構文木の各ノードは identifier octet という 1 バイトで始まり、クラスとタグ番号および constructed というフラグによってノードの種類が表現される。クラスは identifier octet の上位 2 ビット、constructed はその次の 1 ビットである。タグ番号は下位 5 ビットおよび必要ならば後続のバイトで表現される。タグ番号は以下のような形式で記述される。

low-tag-number form 0 から 30 までの番号は identifier octet の下位 5 ビットに埋め込まれる。

high-tag-number form 31 以上の番号は、後続のバイトで表現する。そのことを示すため、identifier octet の下位 5 ビットは 31 とする。後続のバイトはタグ番号を 128 進数として表現して最上位桁から 1 桁を 1 バイトの下位 7 ビットに配置して並べられる。それぞれのバイトの最上位ビットは最後のバイトを除いて 1 として終端を示す。

タグ番号は冗長な表現が可能である。たとえば、high-tag-number form の 128 進数で上位桁に余計な 0 をつけることができる。ファジングではこれを生成する。図 4 に identifier octet とタグ番号を扱うコードを示す。

まず、tokens がバイトのリストであるので、!cons() でこれを消費して最初のバイトを identifier_octet として取り出す。この呼び出しでは左右両方に tokens という同名の変数があるが、これらは異なる変数である。左の tokens は最初のバイトを含んだリストであり、右の tokens は最初のバイトを除いたリストである。divmod(32) で identifier octet を上位 3 ビットと下位 5 ビットに分割する。divmod(2) で上位 3 ビットをさらに上位 2 ビットと下位 1 ビットに分割し、それらがクラスと constructed になる。asn1_parse_tag_number() により必要なら後続のバイトも利用してタグ番号を得る。ここで asn1_parse_tag_number() は冗長な表現も扱い、必要以上に長い場合には tag_extra_bytes が 0 よりも大きくなる。ファジングでは tag_extra_bytes を増やすことにより、asn1_parse_tag_number に冗長な表現を生成させる。

```
backward_fuzzing("premature_end_at_content_length_beginning_octet") { tokens = []; };
```

図 3 途中で途切れた証明書の生成

```
tokens -> !cons() -> identifier_octet, tokens;
identifier_octet -> divmod(32) -> hi_identifier_octet, low_tagnum;
hi_identifier_octet -> divmod(2) -> asn1class, constructed;
low_tagnum, tokens -> asn1_parse_tag_number() -> tagnum, tag_extra_bytes, tokens;
backward_fuzzing("too_long_tagnum_encoding") { tag_extra_bytes = tag_extra_bytes + 4; };
```

図 4 identifier octet とタグ番号を扱うコード

5.3 ファジングの実験結果

証明書をファジングした実験により、以下の結果が得られた。

- テストに用いた証明書を正しく生成する変換の最中には 404 回のファジング機会があった。
- 変換 1 回について 1 回ファジングを適用して間違った証明書を生成して署名の検証を行った。(404 種類, 1.8 分)
- 変換 1 回について 2 回ファジングを適用して間違った証明書を生成して署名の検証を行った。(82405 種類, 7 時間)
- 不適切に署名検証が成功する場合があった。
- 暗号化通信路は確立しなかった。
- テストプログラムは異常終了しなかった。

不適切に署名検証が成功する場合については調査の結果、DER の仕様に反しているものを受けつけてしまうが、脆弱性になるとは考えにくいことがわかった。証明書には、署名の対象になる内容、署名のアルゴリズム、署名自体が含まれている。署名検証が成功してしまうのは、内容以外の部分で整数が冗長に表現されている場合であることがわかった。つまり、署名の対象は 1 バイトでも異なれば確実に検出されるが、それ以外の部分では冗長な整数表現を受け入れてしまっていることになる。冗長な表現は DER の仕様に反している。しかし、署名の対象になる内容は改竄できないので、脆弱性になるとは考えにくい。

6. 関数型言語との比較

Conservation には副作用がなく、その点では関数型言語と似ている。しかし、高階関数は扱えないため、関数型言語ではない。この節では関数型言語との比較を行う。

6.1 高階関数

Conservation にクロージャの導入を検討したが、これは難しいことがわかった。一般にクロージャは自由変数を内部に保存し、呼び出される度にそれらを利用する。しかし、もし自由変数がその呼び出しにおいて消費されるなら、クロージャは 1 回だけしか呼び出せない。これでは `map` などに利用できず、あまり有用でない。

6.2 末尾再帰

関数型言語では末尾再帰を活用することがあるが、Conservation では利用できない。逆方向に実行可能であるために、制御が合流する時には戻れるようにするための条件が必要である。このことは、条件文には合流するところに表明が必要であるだけでなく、`goto` や末尾再帰ができないことも意味する。

`goto A` は `goto` を記述したところから `A` に制御が飛ぶ。逆方向実行が可能であるためには、`A` に制御が到達した時に、そのまま進めるか、`goto` のところに制御が飛ぶか、どちらかが判断可能でなければならない。したがって、`A` のところにその判断を行う条件が必要になる。そのような条件を記述できるようにしないかぎり、`goto` は導入できない。

末尾再帰も同様に導入できない。末尾再帰は実質的には `goto` であり、導入できない理由も同じである。末尾再帰を行う関数を逆方向に動作させられないのはその関数が何段階再帰すればいいかがわからないためである。

7. 関連研究

逆方向に実行できる他の言語として Janus および Prolog について述べる。

7.1 Janus

Janus[1] は決定的に逆方向実行できるが、変数がグローバル変数しかなく、また、データ構造も扱えない。再帰は可能だが、ローカル変数がないため、実用的に利用するのは難しい。これに対し Conservation はローカル変数とデータ構造を扱えるため、再帰下降構文解析を容易に実装できる。

7.2 Prolog

Prolog では述語を成立させる解を探索できるので、述語に対し、値を与える引数と与えない引数を変えることで逆方向実行を実現できる。しかし、これは複数の解が発見される可能性があり、決定的な動作にはならない。なお、Prolog には DCG (Definite Clause Grammar) という言語内 DSL があり、パーザの記述を容易にしている。しかし、構文木の生成については制約が無く、逆方向に決定的に動作できるだけの情報を残すかどうかは個々のプログラムに依存する。これに対し Conservation は決定的に逆方向実行できるプログラムを確実に記述できる。

8. まとめと今後の課題

本論文では逆方向実行可能プログラミング言語 Conservation を提案した。また、Conservation により X.509 証明書のエンコーダ・デコーダを記述し、ファジングにも利用できることを示した。

X.509 証明書については、今回 ASN.1 DER のレベルで証明書を扱い、証明書のために定義された構造 [5] は利用しなかった。今後、証明書の構造を意識した構文木を扱うようにしたい。

また、ファジングは用意したひとつのサーバ証明書に対してファジングを行った。したがって、テストはその証明書で使われる構文の周辺に対して行われており、仕様に対して網羅的にテストしているとはいえない。そこで、そのような元データを用いずに網羅的に証明書を生成し、ファ

ジングを行うことを考えたい。エンコーダ・デコーダは構文の仕様なので、構文から導出可能な文を枚挙することは可能なはずである。

さらに、今回は証明書の中の署名を再利用したが、間違った内容に対して署名を行って埋め込むことも考えたい。この場合、デコーダで構文木を生成する時には署名を (検証して) 取り除き、エンコーダで証明書を生成する時には署名を生成することになる。

参考文献

[1] Tetsuo, Yokoyama, and Robert Glck: A reversible programming language and its invertible self-interpreter, Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. ACM (2007).

[2] Burton S. Kaliski Jr.: A Layman's Guide to a Subset of ASN.1, BER, and DER, An RSA Laboratories Technical Note (1993) available from ⟨http://luca.ntop.org/Teaching/Appunti/asn1.html⟩.

[3] Michael Sutton, Adam Greene, Pedram Amini: Fuzzing — Brute Force Vulnerability Discovery, Addison-Wesley (2007).

[4] T. Dierks, E. Rescorla: The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246 (2008).

[5] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC 5280 (2008).

[6] X.509 : Information technology — Open systems interconnection — The Directory: Public-key and attribute certificate frameworks ITU-T (2008).

[7] X.680 : Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation, ITU-T (2008).

[8] X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), ITU-T (2008).

[9] OpenSSL: The Open Source toolkit for SSL/TLS, available from ⟨http://www.openssl.org/⟩

付　　録

A.1 両方向プリミティブ関数

本論文で使用している両方向プリミティブ関数を以下に示す。

- x `->` `equal()` `->` y
  恒等関数。順方向実行では x を消費してその値を y に束縛する。
- `bind(`e`)` `->` v
  順方向実行では、e を v に束縛する。逆方向実行では、v と e が等しければ成功し、異なっていれば失敗する。
- `guard(`b`)`
  単方向引数 b が偽の場合には失敗する。
- n `->` `divmod(`m`)` `->` q`,` r
  順方向実行では n, m を受け取り、q, r を束縛する。逆方向実行では q, r, m を受け取り、n を束縛する。m, n, q, r は整数であり、$n = qm + r$, $0 < m$, $0 \leq r < m$ という関係が成り立つ。
- x `->` `add(`y`)` `->` z
  $x + y = z$ が成り立つように結果を決定する。順方向実行では $x + y$ を z に束縛する。逆方向実行では $z - y$ を x に束縛する。
- x `->` `sub(`y`)` `->` z
  $x - y = z$ が成り立つように結果を決定する。順方向実行では $x - y$ を z に束縛する。逆方向実行では $z + y$ を x に束縛する。なお、これは x `->` `!add(`y`)` `->` z と等価である。
- h`,` t `->` `cons()` `->` l
  順方向実行では h,t を受け取り、頭部が h、尾部が t な cons セルを生成して l に束縛する。逆方向実行では l を cons セルとして頭部と尾部に分解し、h, t に束縛する。逆方向実行において l が cons セルでなかった場合には失敗する。
- l_1`,` l_2 `->` `concat()` `->` n`,` l_0
  順方向実行では、ふたつのリスト l_1, l_2 を連結し、l_1 の長さを n に、連結したリストを l_0 に束縛する。逆方向実行では、リスト l_0 を先頭 n 要素とそれ以降に分割し、前半を l_1、後半を l_2 に束縛する。
- n`,` s `->` `sint_be()` `->` l
  順方向実行では、整数 n を s バイトのビッグエンディアンな符号付整数にエンコードし、バイト (整数) のリスト l に束縛する。逆方向実行ではバイトのリスト l をビッグエンディアンな符号付整数と解釈し、その値とリストの長さを n, s に束縛する。
- n`,` s `->` `uint_be()` `->` l
  順方向実行では、整数 n を s バイトのビッグエンディアンな符号無整数にエンコードし、バイト (整数) のリスト l に束縛する。逆方向実行ではバイトのリスト l をビッグエンディアンな符号無整数と解釈し、その値とリストの長さを n, s に束縛する。
- $t_1, \ldots$ `->` `mknode(`n`,` d`)` `->` t_0
  順方向では構文木のノードを作り、逆方向では分解する。ノードの種類は非終端記号の名前 n と導出規則の名前 d で指定する。(代数的データ型と考えれば、型の名前とコンストラクタの名前ともみなせる。) 順方向では、部分構文木を $t_1, \ldots$ で与え、それらを子として持つノードが t_0 として得られる。ノードは非終端記号と導出規則の名前も記録しており、逆方向では単方向引数 n, d とノード t_0 で名前が一致しなければ失敗する。`mknode` は本論文で唯一の可変長引数関数であり、本論文で示した関数定義の記法では定義できない。関数定義の記法を拡張して定義可能にすることもできるが、論旨に影響しないので省略した。
- *asn1class*`,` *tagnum* `->` `asn1_tagname()` `->` *tag*
  ASN.1 のクラス番号とタグ番号と、人間にわかりやすいタグの名前を変換する。順方向では前者から後者、逆方向では後者から前者に変換する。
- *contents* `->` `object_identifier()` `->` *block*
  ASN.1 のオブジェクト識別子の名前と、対応するバイトのリストを変換する。順方向では前者から後者、逆方向では後者から前者に変換する。

- *contents* `->` `utctime()` `->` *block*
  ISO 8601 形式で表現した時刻の文字列と ASN.1 の UTCTime に対応するバイトのリストを変換する。順方向では前者から後者、逆方向では後者から前者に変換する。
- *contents* `->` `ia5string()` `->` *block*
  文字列と ASN.1 の IA5String に対応するバイトのリストを変換する。順方向では前者から後者、逆方向では後者から前者に変換する。
- *contents* `->` `printable_string()` `->` *block*
  文字列と ASN.1 の PrintableString に対応するバイトのリストを変換する。順方向では前者から後者、逆方向では後者から前者に変換する。
- *contents* `->` `utf8string()` `->` *block*
  文字列と ASN.1 の UTF8String に対応するバイトのリストを変換する。順方向では前者から後者、逆方向では後者から前者に変換する。

A.2 単方向プリミティブ関数

本論文で使用している単方向プリミティブ関数を以下に示す。

- x `==` y
  x と y が等しいときに真を返す。
- x `!=` y
  x と y が等しくないときに真を返す。
- x `<` y
  x よりも y が大きいときに真を返す。
- x `<=` y
  x よりも y が大きいか等しいときに真を返す。
- x `&&` y
  x と y が両方とも真であるときに真を返す。
- `num_digits(`*base*`,` *num*`)`
  num を *base* 進数として表現したときの桁数を返す。
- `asn1_integer_minlen(`*num*`)`
  整数 *num* を ASN.1 の INTEGER という型で DER として表現した場合のバイト数を返す。これは 2 の補数表現でのバイト数である。ただし、*num* が 0 ならば 1 を返す。
- `appropriate_content_length_bytesize(`*length*`)`
  ASN.1 DER において内容の長さ *length* を表現するためのバイト数のうち、後続のバイトの長さを返す。これは *length* < 128 ならば 0、そうでなければ *length* を 256 進数で表現したときの桁数である。内容の長さは、127 以下なら最初のバイトだけで表現し、そうでなければ最初のバイトが後続のバイト数を表現する。後続のバイトがある場合には、それをビッグエンディアンの 256 進数として解釈したものが内容の長さになるという形式である。

A.3 ASN.1 DER エンコーダ・デコーダ

```
define tokens -> asn1_parse_objs() -> tree
  cond
  when { guard(tokens == []); } {
      tokens -> asn1_parse_obj() -> head, tokens;
      tokens -> asn1_parse_objs() -> tail;
    } { head, tail -> mknode("objs", "pair") -> tree; }
  when { tokens -> !bind([]); } {
    } { mknode("objs", "empty") -> tree; }
  end
end
```

```
define tokens -> asn1_parse_obj() -> tree, tokens
  tokens -> !cons() -> identifier_octet, tokens;
  identifier_octet -> divmod(32) -> hi_identifier_octet, low_tagnum;
  hi_identifier_octet -> divmod(2) -> asn1class, constructed;
  low_tagnum, tokens -> asn1_parse_tag_number() -> tagnum, tag_extra_bytes, tokens;
  backward_fuzzing("too_long_tagnum_encoding") { tag_extra_bytes = tag_extra_bytes + 4; };
  asn1class, tagnum -> asn1_tagname() -> tag;
  tokens -> asn1_parse_content_length() -> content_length, content_length_extra_bytes, tokens;
  backward_fuzzing("too_long_length_encoding") { content_length_extra_bytes = content_length_extra_bytes + 1;
  backward_fuzzing("premature_end_at_content") { tokens = []; } if content_length != 0;
  content_length, tokens -> !concat() -> block, tokens;
  cond
  when { guard(constructed == 0); } {
      block -> asn1_parse_primitive(tag) -> contents;
    } { guard(constructed == 0); }
  when { guard(constructed == 1); } {
      block -> asn1_parse_objs() -> contents;
    } guard(constructed == 1); }
  end
  constructed, tag_extra_bytes, tag, content_length_extra_bytes, contents -> mknode("obj", "default") -> tree
end

define block -> asn1_parse_primitive(tag) -> contents
  cond
  when { guard(tag == "INTEGER"); } {
      block -> !sint_be() -> num, size;
      size -> sub(asn1_integer_minlen(num)) -> integer_extra_bytes;
      backward_fuzzing(too_long_integer_encoding") { integer_extra_bytes = integer_extra_bytes + 4; };
      bind([num, integer_extra_bytes]) -> contents;
    } { guard(tag == "INTEGER"); }
  when { guard(tag == "UTF8String"); } {
      block -> !utf8string() -> contents;
    } { guard(tag == "UTF8String"); }
  when { guard(tag == "IA5String"); } {
      block -> !ia5string() -> contents;
    } { guard(tag == "IA5String"); }
  when { guard(tag == "PrintableString"); } {
      block -> !printable_string() -> contents;
    } { guard(tag == "PrintableString"); }
  when { guard(tag == "OBJECT_IDENTIFIER"); } {
      block -> !object_identifier() -> contents;
    } { guard(tag == "OBJECT_IDENTIFIER"); }
  when { guard(tag == "UTCTime"); } {
      block -> !utctime() -> contents;
    } { guard(tag == "UTCTime"); }
  when otherwise {
      block -> equal() -> contents;
```

```
    } otherwise
  end;
end

define low_tagnum, tokens -> asn1_parse_tag_number() -> tagnum, extra_bytes, tokens
  cond
  when { guard(low_tagnum != 31); } {
      low_tagnum -> equal() -> tagnum;
    } { bind(0) -> hightagsize; }
  when { low_tagnum -> !bind(31); } {
      bind(0) -> i;
      bind(0) -> j;
      i, j, tokens -> asn1_parse_base128() -> tagnum, hightagsize, tokens;
    } { guard(hightagsize != 0); }
  end;
  hightagsize -> sub(num_digits(128, tagnum)) -> extra_bytes;
end

define hi_tagnum, len, tokens -> asn1_parse_base128() -> tagnum, len, tokens
  backward_fuzzing("premature_end_at_hi_tag_number_octet") { tokens = []; };
  tokens -> !cons() -> byte, tokens;
  cond
  when { guard(0 <= byte && byte < 128); } {
      hi_tagnum, byte -> !divmod(128) -> tagnum;
      len -> add(1) -> len;
    } { guard(len == 1); };
  when { guard(128 <= byte && byte < 256); } {
      byte -> divmod(128) -> one, d;
      one -> !bind(1);
      hi_tagnum, d -> !divmod(128) -> tagnum1;
      tagnum1, len, tokens -> asn1_parse_base128() -> tagnum, len, tokens;
      len -> add(1) -> len;
    } { guard(2 <= len); }
  end
end

define tokens -> asn1_parse_content_length() -> length, content_length_extra_bytes, tokens
  backward_fuzzing("premature_end_at_content_length_beginning_octet") { tokens = []; };
  tokens -> !cons() -> byte, tokens;
  byte -> divmod(128) -> msb, lowbits;
  cond
  when { msb -> !bind(0); } {
      lowbits -> equal() -> length;
    } { bind(0) -> content_length_bytesize; };
  when { msb -> !bind(1); guard(lowbits != 0); } {
      backward_fuzzing("premature_end_at_long_content_length") { tokens = []; } if lowbits != 0;
      lowbits, tokens -> !concat() -> block, tokens;
      block -> !uint_be() -> length, content_length_bytesize;
```

```
    } { guard(0 < content_length_bytesize); }
  end;
  backward_fuzzing("too_long_length_encoding_max") { content_length_bytesize = 127; };
  content_length_bytesize -> sub(appropriate_content_length_bytesize(length)) -> content_length_extra_bytes;
end
```