ProVerifによるTLS1.3ハンドシェイクプロトコルの形式検証

荒井 研一* 渡辺 大† 櫻田 英樹‡

* 東京理科大学
278-8510 千葉県野田市山崎 2641
k.arai@rs.tus.ac.jp

† 株式会社日立製作所 研究開発グループ
244-0817 神奈川県横浜市戸塚区吉田町 292
dai.watanabe.td@hitachi.com

‡ 日本電信電話株式会社 NTT コミュニケーション科学基礎研究所
243-0198 神奈川県厚木市森の里若宮 3-1
sakurada.hideki@lab.ntt.co.jp

あらまし ProVerif は Blanchet らが開発した形式モデルでの暗号プロトコルの自動検証ツールであり，暗号プロトコルで要求される秘匿や認証などの安全性を検証可能である．一方，TLS(Transport Layer Security) はインターネット上で安全な通信を提供する暗号プロトコルであり，ハンドシェイクプロトコルは相手認証及び安全な通信を確立するために必要なセキュリティパラメータのネゴシエーションを行うプロトコルである．本稿では，ProVerif を用いて TLS1.3(TLS protocol version 1.3) ハンドシェイクプロトコルを形式的に記述し，安全性検証を行う．

Formal Verification of TLS 1.3 Handshake Protocol Using ProVerif

Kenichi Arai* Dai Watanabe† Hideki Sakurada‡

* Tokyo University of Science
2641 Yamazaki, Noda, Chiba 278-8510, JAPAN
k.arai@rs.tus.ac.jp

†Hitachi, Ltd., Research & Development Group
292 Yoshida-cho, Totsuka-ku, Yokohama, Kanagawa, 244-0817, JAPAN
dai.watanabe.td@hitachi.com

‡NTT Communication Science Laboratories, NTT Corporation
3-1 Morinosato Wakamiya, Atsugi, Kanagawa 243-0198, JAPAN
sakurada.hideki@lab.ntt.co.jp

Abstract ProVerif is an automatic cryptographic protocol verifier based on the formal model developed by Blanchet et al. This verifier can verify the properties of secrecy and authentication, etc. On the other hand, TLS (Transport Layer Security) is a cryptographic protocol that provides communications security over the Internet. The handshake protocol is a protocol that provides a means of authentication and the negotiation of security parameters in TLS. In this paper, we introduce our formalization of the TLS 1.3 (TLS protocol version 1.3) handshake protocol and verify its security using ProVerif.

1 はじめに

ProVerif[1] は Blanchet らが開発した形式モデルでの暗号プロトコルの自動検証ツールであり，暗号プロコルで要求される秘匿や認証などの安全性を検証可能である．ProVerif はさまざまな暗号プロトコルの検証が可能であり，多くの暗号プロトコルに対して脆弱性を発見することに成功している．

一方，TLS(Transport Layer Security) はインターネット上で安全な通信を提供する暗号プロトコルであり，具体的には通信の暗号化 (秘匿)，データ完全性の確保，相手認証 (サーバ認証，場合によってはクライアント認証) の機能を提供する暗号プロトコルである [2]．TLS は，米 Netscape 社によって開発された SSL3.0(Secure Sockets Layer version 3.0) [3] をベースとしている．SSL は，主に Web アクセス (HTTP) において安全な通信を確保するために利用される暗号プロトコルである．1999 年にインターネット技術の標準化推進団体である IETF(Internet Engineering Task Force) は，SSL3.0 をベースとして TLS Protocol Version 1.0 (TLS1.0) を策定した [4]．その後，TLS1.0 の安全性強化を図るため，2006 年に TLS1.1 が策定された [5]．TLS1.1 では，Vaudenay[6] や Yau ら [7] など，暗号利用モードの一つである CBC モードの脆弱性に対する指摘を受け対処が行われた．さらに，2008 年に TLS1.2 が策定された [8]．TLS1.2 は，より安全な暗号スイート (ハッシュ関数 SHA-256 及び SHA-384 の追加，暗号利用モードとして CCM[9] 及び GCM[10] といった関連データ付き認証付暗号 (AEAD: Authenticated Encryption with Associated Data) [11] の追加など) を選択可能となった．現在は，IETF において TLS1.3 の標準化の議論が進められており，ドラフトが頻繁に更新されている [1]．TLS1.3 では，ハンドシェイクメッセージのできる限りの暗号化及び見直しが行われ，暗号利用モードとして CBC モードが廃止され AEAD の選択が必須となり，鍵交換においては Perfect Forward Secrecy(PFS) を有する Ephemeral DH(DHE), ECDH(ECDHE) などの Diffie-Hellman 鍵交換の選択が必須となり，データ圧縮が非サポートとなった．RFC(Request For Comment) の発行前ではあるが，[13] などドラフトの安全性に対して既に活発に研究が行われている．

本稿ではドラフトが頻繁に更新されている TLS1.3 の安全性評価の一環として，draft-06[14] におけるハンドシェイクプロトコルについて ProVerif を用いて形式的に記述し，安全性検証を行う．なお，ハンドシェイクプロトコルは相手認証及び安全な通信 (暗号化通信) を確立するために必要なセキュリティパラメータのネゴシエーションを行うプロトコルである．ハンドシェイクプロトコル，なかでも full ハンドシェイクはドラフトの更新毎に多少の変更が加えられているが，大枠の変更はほとんどない．よって，安全性評価の一環として draft-06 における full ハンドシェイクについて安全性検証を行う．本研究は，形式検証ツールを用いた評価技術の普及を目指すものである．

2 ProVerif

ProVerif は通信路に流れるメッセージを記号に理想化して暗号プロトコルの検証を行う形式手法 (いわゆる，Dolev-Yao モデル [15]) を用いて，暗号プロトコルの安全性を検証している．ProVerif では，Blanchet らのプロセス計算 (拡張 π 計算 [16]) の構文規則 (または，Horn 節 [17]) を用いて暗号プロトコル (暗号プリミティブ及び安全性要件を含む) を記述する．本稿で用いた Blanchet らのプロセス計算の構文規則を図 1 に示す．

ProVerif は，これらの構文規則を用いて記述された暗号プロトコルを Horn 節 (の集合) に自動変換する．なお，一般的に節は $\neg F_1 \vee \ldots \vee \neg F_m \vee G_1 \vee \ldots \vee G_n$ の形をした否定及び肯定のリテラルの選言からなる．このとき，n の値が 1 以下のものを Horn 節と呼ぶ．よって，Horn 節は含意記号 ($\Rightarrow$) を用いると $F_1 \wedge \ldots \wedge F_m \Rightarrow G_1$ のように表すことができる．ここで，F 及び G は述語記号によって表現される事実である．暗号プロトコルの Horn 節表現に用いられる主な述語記号は **attacker** であり，**attacker**(M) は，

[1] TLS1.3 は draft-08 まで更新されている (2015 年 8 月 23 日現在)．最新版の詳細は，[12] を参照されたい．

$M, N ::=$	項
a, b, c, k, m, n, s	定数
x, y, z	変数
$(M_1, ..., M_k)$	組
$h(M_1, ..., M_k)$	コンストラクタ/ デストラクタ関数
$M = N$	項の等式
$M <> N$	項の不等式
$M \&\& M$	連言
$P, Q ::=$	プロセス
0	何もしない
$P \mid Q$	並列合成 (並行実行)
$!P$	複製
new $n : t; P$	定数生成
in$(M, x : t); P$	メッセージ入力 (受信)
out$(M, N); P$	メッセージ出力 (送信)
if M **then** P **else** Q	条件分岐
let $x = M$ **in** P **else** Q	項の評価
$R(M_1, ..., M_k)$	マクロ
event $e(M_1, \ldots, M_n); P$	(事前/事後) イベント

図 1: 構文規則

“攻撃者は M を得ることができる” という事実を意味する．この述語記号によって攻撃者及びプロトコル参加者の動作をモデル化することができる．自動変換により得られた Horn 節の集合は，初期節と呼ばれ，攻撃者の計算能力，攻撃者の初期知識及びプロトコルを Horn 節表現したものから構成される．ProVerif は，これら初期節を用いて導出 (Resolution) アルゴリズムを実行することにより，安全性要件に反する事実がホーン節の集合より導出できるかどうかを検証する．安全性要件に反する事実を Horn 節の集合より導出できないとき，その暗号プロトコルは安全性要件を満たす．一方，安全性要件に反する事実を Horn 節の集合より導出できたとき，その暗号プロトコルは攻撃可能であるとし，攻撃手順を表示する．導出アルゴリズムの詳細は，[17, 18] を参照されたい．

2.1 ProVerif による検証

ProVerif は，秘匿 [18] 及び認証 [19] の安全性要件を検証可能である．本節では，ProVerif による秘匿及び認証の検証について説明する．

2.1.1 秘匿

ProVerif による秘匿の検証は，攻撃者が秘密情報 (秘匿したい情報) を入手可能か否かを検証することにより行われる．秘匿の検証は ProVerif における最も基本的な機能である．

2.1.2 認証

認証 (いわゆる，相手認証) が成立しているということは，送信者を特定するための事実を受信者が正しく受け取っているということである．認証は対応表明 [19] を用いて定義される．ProVerif での認証は，認証用パラメータが生成されたときを “事前イベント”，プロトコルが完了したときを “事後イベント” として定義し，事後イベントが実行された際に，それに対応する事前イベントが必ず存在しているか否かを検証することにより行われる．対応する事前イベントが必ず存在していれば，同じ認証用パラメータを用いて通信を行った送信者が存在するといえる．一方，事前イベントが存在しない場合は，正規の手順でプロトコルを実行せずにプロトコルを完了 (認証成立) できたことを意味する．すなわち，なりすましが可能であることを意味する．さらに，ProVerif は再生攻撃の可能性についても検証可能である．

3 TLS プロトコル

TLS はトランスポート層とアプリケーション層との間に位置する暗号プロトコルであり，それらの間に位置することでアプリケーション層における特定のプロトコルに依存することなく，通信の暗号化 (秘匿)，データ完全性の確保，相手認証 (サーバ認証，場合によってはクライアント認証) といった機能を提供することができる．

TLS プロトコルでは，安全な通信 (暗号化通信) を開始するにあたって，ハンドシェイクプロトコルが実行される．ハンドシェイクプロトコルでは，クライアントとサーバが暗号化通信を行うために利用する暗号スイート (暗号アルゴリズム) 及びプロトコルバージョンを決定し，サーバ証明書 (X.509 証明書 [20]) を用いてサーバ認証を行い，本セッションで利用するセキュリティ

パラメータをクライアントとサーバで共有する．これら一連の動作を行うのがハンドシェイクプロトコルである．なお，サーバ証明書だけでなくクライアント証明書を用いることにより，クライアント認証を行うことも可能である．この場合は相互認証となる．TLS においては，クライアント認証はオプションとなっている．ハンドシェイクプロトコル完了後，クライアントとサーバは，共有したセキュリティパラメータより生成される鍵 (及び IV) を用いて送受信するアプリケーションデータを暗号化し，かつデータ完全性を確保することにより安全な通信を実現している．TLS プロトコルの詳細は，[2, 14] を参照されたい．

3.1 ハンドシェイクプロトコル

本節では，TLS1.3 draft-06 におけるハンドシェイクプロトコル，なかでも full ハンドシェイクについて説明する．その他のハンドシェイクについての詳細は，[14] を参照されたい．なお，TLS1.3 ではハンドシェイクメッセージのできる限りの暗号化が行われ，暗号利用モードとして AEAD の選択が必須となり，鍵交換においては PFS を有する Ephemeral(一時的) な Diffie-Hellman(DH) 鍵交換の選択が必須となり，データ圧縮が非サポートとなっている．

full ハンドシェイクのメッセージフローを図 2 に示す．

```
     Client                                  Server
-------------------------------------------------------
ClientHello
ClientKeyShare       -------->
                                               ServerHello
                                            ServerKeyShare
                                  {EncryptedExtensions*}
                                         {Certificate*}
                                   {CertificateRequest*}
                                    {CertificateVerify*}
                     <--------                 {Finished}
{Certificate*}
{CertificateVerify*}
{Finished}           -------->
[Application Data]   <------->     [Application Data]
-------------------------------------------------------
```

図 2: full ハンドシェイクのメッセージフロー

ここで，∗はオプション項目であり省略可能である．さらに，{ } は handshake master secret による暗号化，[] は master secret から生成される鍵による暗号化を表している．

続いて，図 2 における処理手順を以下に示す．なお，本稿では一般的な TLS の利用法として，サーバ認証のみを行うケースを形式化する．すなわち，図 2 に含まれるハンドシェイクメッセージのうち，サーバ側の **EncryptedExtensions** 及び **CertificateRequest**，クライアント側の **Certificate** 及び **CertificateVerify** は本形式化では省略した．省略項目についての詳細は，[14] を参照されたい．

1-1. クライアントは，プロトコルバージョン，(クライアント) 乱数，セッション ID，暗号スイートの一覧，及び圧縮アルゴリズム (“null(0)”) を **ClientHello** として生成する．

1-2. クライアントは，一時的な DH 鍵を生成するための DH 公開値を **ClientKeyShare** として生成する．

1-3. クライアントは，**ClientHello** 及び **ClientKeyShare** をサーバに送信する．

2-1. **ClientHello** 及び **ClientKeyShare** を受信したサーバは，**ClientHello** より利用する暗号スイート及びプロトコルバージョンを決定する．

2-2. サーバはプロトコルバージョン，(サーバ) 乱数，セッション ID，及び暗号スイートを **ServerHello** として生成する．

2-3. サーバは一時的な DH 鍵を生成するための DH 公開値を **ServerKeyShare** として生成し，**ClientKeyShare** より pre master secret(以降，pMS)，すなわち一時的な DH 鍵を生成する．さらに，pMS，ラベル (“handshake master secret”)，及び今までに送受信したハンドシェイクメッセージ (の連結) のハッシュ値 (session hash) より PRF (擬似ランダム関数) を用いて handshake master secret(以降，hsMS) を生成する．

2-4. サーバは，サーバ証明書 **Certificate** を hsMS を用いて (共通鍵暗号方式で) 暗号化する．

2-5. サーバは今までに送受信したハンドシェイクメッセージのハッシュ値に対する署名 **CertificateVerify** を (ディジタル署名方式

を用いて) 生成し，その署名を hsMS を用いて暗号化する．

2-6. サーバは pMS，ラベル (“server finished”)，及び今までに送受信したハンドシェイクメッセージのハッシュ値より PRF を用いてハンドシェイク終了通知 **Finished** を生成し，そのハンドシェイク終了通知を hsMS を用いて暗号化する．

2-7. サーバは hsMS，ラベル (“extended master secret”)，及び今までに送受信したハンドシェイクメッセージのハッシュ値 (session hash) より PRF を用いて master secret(以降，MS) を生成する．

2-8. サーバは **ServerHello** 及び **ServerKeyShare**，暗号化された **Certificate**，**CertificateVerify** 及び **Finished** をクライアントに送信する．

3-1. **ServerHello** 及び **ServerKeyShare**，暗号化された **Certificate**，**CertificateVerify** 及び **Finished** を受信したクライアントは，利用する暗号スイート及びプロトコルバージョンを確認する (クライアント及びサーバ双方の合意)．

3-2. クライアントは，**ServerKeyShare** より pMS を生成する．さらに，サーバと同様の方法で hsMS を生成する．

3-3. 暗号化された **Certificate**，**CertificateVerify**，及び **Finished** を hsMS を用いて復号する．

3-4. 復号した **Certificate** 及び **CertificateVerify** を用いて証明書検証及び署名検証を行う．証明書検証が成功するとその証明書がサーバのものであることを確認でき，署名検証が成功すると通信相手が確かにサーバであることを確認できる．

3-5. クライアントは復号した **Finished** を確認する．すなわち，クライアント側でも **Finished** を生成し，復号した **Finished** と比較する (ハンドシェイクメッセージが改ざんされていないことの確認)．

3-6. クライアントは，サーバと同様の方法で MS を生成する．

3-7. クライアントは hsMS，ラベル (“client finished”)，及び今までのハンドシェイクメッセージのハッシュ値より PRF を用いてハンドシェイク終了通知 **Finished** を生成し，そのハンドシェイク終了通知を hsMS を用いて暗号化する．

3-8. クライアントは，暗号化された **Finished** をサーバに送信する．

4-1. 暗号化された **Finished** を受信したサーバは，暗号化された **Finished** を hsMS を用いて復号する．

4-2. サーバは，復号した **Finished** を確認する．

5-1. クライアント及びサーバ双方は，双方で共有している MS，ラベル (“key expansion”)，及びサーバ乱数とクライアント乱数 (の連結) より PRF を用いて key_block を生成する．さらに，key_block を

client_write_key，server_write_key，
client_write_IV，server_write_IV

に分割する．

5-2. クライアント及びサーバ双方は，シーケンス番号，型 (type)，及びレコードバージョン (の連結) より additional_data を生成する．さらに，シーケンス番号及び client_write_IV (もしくは，server_write_IV) より XOR (排他的論理和) を用いてレコードノンスを生成する．

5-3. クライアント及びサーバ双方は，client_write_key(もしくは，server_write_key)，**Application Data**，レコードノンス，及び additional_data より AEAD を用いて暗号文を生成し，その暗号文，レコードノンス，及び additional_data を送信する．AEAD を用いて **Application Data** を暗号化し，かつデータ完全性を確保することにより安全な通信を実現している．

4 full ハンドシェイクの形式化

本節では，TLS1.3 draft-06 における full ハンドシェイクの形式化について説明する．なお，本形式化はサーバ認証のみを行うケースを形式化している．また，各項の型はすべて予約語である bitstring 型としている．これは，型の不一致などの攻撃の発見も考慮に入れているためである．

ProVerif のコードは宣言部及びプロセス部より構成される．宣言部は検証対象の暗号プロト

コルで用いられる関数 (コンストラクタ/デストラクタ関数)，性質，及び安全性要件などを記述し，プロセス部は検証対象の暗号プロトコルそのものを記述する．

4.1 宣言部

full ハンドシェイクの形式化における宣言部のコードを以下に示す．

```
 1  free c:channel.
 2 (* label *)
 3  const HANDSHAKE_MASTER_SECRET:bitstring [data].
 4  const EXTENDED_MASTER_SECRET:bitstring [data].
 5  const SERVER_FINISHED:bitstring [data].
 6  const CLIENT_FINISHED:bitstring [data].
 7  const KEY_EXPANSION:bitstring [data].
 8 (* Host Information *)
 9  free HostInfoC, HostInfoS, HostInfoCA:bitstring.
10 (* Hash *)
11  fun H(bitstring):bitstring.
12 (* PRF *) (* P_Hash *)
13  fun P_hash(bitstring,bitstring):bitstring.
14   reduc forall secret:bitstring,label:bitstring,seed:bitstring;
15    PRF(secret,label,seed) = P_hash(secret,(label,seed)).
16 (* Sign *)
17  fun sign(bitstring,bitstring):bitstring.
18  fun pk(bitstring):bitstring.
19   reduc forall x:bitstring,sk:bitstring;
20    verify(sign(x,sk),x,pk(sk)) = true.
21 (* Shared key encryption *)
22  fun enc(bitstring,bitstring):bitstring.
23   reduc forall x:bitstring,sk:bitstring; dec(enc(x,sk),sk) = x.
24 (* Diffie-Hellman *)
25  const g: bitstring.
26  fun exp(bitstring,bitstring):bitstring.
27   equation forall x:bitstring,y:bitstring;
28    exp(exp(g,x),y) = exp(exp(g,y),x).
29 (* XOR *)
30  fun xor(bitstring,bitstring):bitstring.
31   equation forall x:bitstring,y:bitstring; xor(xor(x,y),y) = x.
32   equation forall x:bitstring; xor(x,xor(x,x)) = x.
33   equation forall x:bitstring; xor(xor(x,x),x) = x.
34   equation forall x:bitstring,y:bitstring; xor(y,xor(x,x)) = y.
35   equation forall x:bitstring,y:bitstring;
36    xor(xor(x,y),xor(x,x)) = xor(x,y).
37   equation forall x:bitstring,y:bitstring;
38    xor(xor(x,y),xor(y,y)) = xor(x,y).
39 (* Key_block partition *)
40  fun key_block_pt_client_write_key(bitstring): bitstring.
41  fun key_block_pt_server_write_key(bitstring): bitstring.
42  fun key_block_pt_client_write_IV(bitstring): bitstring.
43  fun key_block_pt_server_write_IV(bitstring): bitstring.
44 (* AEAD: Authenticated Encryption with Associated Data *)
45  (* Encryption scheme *)
46  fun encrypt(bitstring,bitstring,bitstring): bitstring.
47   reduc forall x:bitstring,k:bitstring,r:bitstring;
48    decrypt(encrypt(x,k,r),k) = x.
49  (* MAC *)
50  fun mac(bitstring,bitstring): bitstring.
51   reduc forall x:bitstring,k:bitstring;
52    verify_mac(mac(x,k),x,k) = true.
53  (* Encrypt-then-MAC (generic composition) *)
54  reduc forall k:bitstring,n:bitstring,p:bitstring,ad:bitstring;
55   AEAD_Encrypt(k,n,p,ad) =
56    (encrypt(p,k,n),mac((n,ad,encrypt(p,k,n)),k)).
57  reduc forall k:bitstring,n:bitstring,p:bitstring,ad:bitstring;
58   AEAD_Decrypt(k,n,(encrypt(p,k,n),
59    mac((n,ad,encrypt(p,k,n)),k)),ad) = p.
60 (* queries - secrecy *)
61  free AppDataClient: bitstring [private].
62   query attacker(AppDataClient).
63 (* queries - authentication *)
64  event endClient(bitstring,bitstring,bitstring,bitstring).
65  event beginClient(bitstring,bitstring,bitstring,bitstring).
66  query s:bitstring,t:bitstring,u:bitstring,v: bitstring;
67   inj-event(endClient(s,t,u,v)) ==> inj-event(beginClient(s,t,u,v)).
```

ここで，(* *) はコメントアウトを意味し，1 行目は公開チャネル，3〜7 行目は full ハンドシェイで用いられるラベル，9 行目は証明書に用いられるサーバ，クライアント，認証局 (CA) の情報の宣言を行っている．

11〜59 行目では full ハンドシェイクで用いられる暗号プリミティブの形式化を行っている．具体的には，11 行目はハッシュ関数，13〜15 行目は擬似ランダム関数 (PRF)，17〜20 行目はディジタル署名方式，22〜23 行目は共通鍵暗号方式，25〜28 行目は DH 鍵共有，30〜38 行目は XOR(排他的論理和)，40〜43 はパーティション関数，46〜59 行目は AEAD の形式化を行っている．DH 鍵共有については，有限体上の DH 鍵共有を形式化している．パーティション関数は，key_block から各鍵を抽出する関数となっている．AEAD の形式化については，[11] の Generic Composition における Encrypt-then-MAC を参考にして形式化を行っている．なお，全ての暗号プリミティブは理想的なものとして形式化している．

61〜67 行目では検証したい安全性要件についての宣言を行っている．61〜62 行目は秘匿の検証を行うための宣言，64〜67 行目は認証の検証を行うための宣言を行っている．秘匿の検証については，クライアントと攻撃者の間で鍵共有を成立させることができるかを検証している (アプリケーションデータの秘匿性)．認証の検証については，攻撃者が (Client に対して) サーバになりすますことができるかを検証している．なお，本形式化はサーバ認証のみを行うケースを形式化しているためクライアント認証に関連する安全性検証は行っていない．

4.2 プロセス部

プロセス部では，サーバ，クライアント，及び認証局の各処理手順を逐次的に記述する．full ハンドシェイクの形式化におけるプロセス部のコードを以下に示す．

```
 1 (* Client process *)
 2  let processClient(pkCA:bitstring) =
 3 (* setup *)
 4  in(c, (client_version:bitstring,client_session_id:bitstring,
 5   client_cipher_suites:bitstring,compression_methods:bitstring,
 6   client_seq_num:bitstring,client_type:bitstring,
 7   client_record_version: bitstring));
 8 (* ClientHello *)
 9  new client_random:bitstring;
10  let client_hello = (client_version,client_random,
11   client_session_id,client_cipher_suites,compression_methods) in
12 (* ClientKeyShare *)
13  new X:bitstring;
14  let client_key_share = exp(g,X) in
15 (* Client Output *)
```

```
 out(c, (client_hello,client_key_share));
(* Client Input *)
 in(c, (server_hello:bitstring,server_key_share:bitstring,
  enc_server_certificate:bitstring,
  enc_server_certificate_verify:bitstring,
  enc_server_finished:bitstring));
 let (server_version:bitstring,server_random: bitstring,
  server_session_id:bitstring,server_cipher_suites:bitstring) =
   server_hello in
 if server_version = client_version &&
  server_cipher_suites = client_cipher_suites then
(* pre_master_secret *)
 let pre_master_secret = exp(server_key_share,X) in
(* session_hash *)
 let session_hash = H((client_hello,client_key_share,
  server_hello,server_key_share)) in
(* hs_master_secret *)
 let hs_master_secret = PRF(pre_master_secret,
  HANDSHAKE_MASTER_SECRET,session_hash) in
(* decryption *)
 let server_certificate =
  dec(enc_server_certificate,hs_master_secret) in
 let server_certificate_verify =
  dec(enc_server_certificate_verify,hs_master_secret) in
  let server_finished = dec(enc_server_finished,hs_master_secret) in
(* check *)
 let (=HostInfoCA,=HostInfoS,pkS:bitstring,signCA_pkS:bitstring) =
  server_certificate in
 if verify(signCA_pkS,H((HostInfoCA,HostInfoS,pkS)),pkCA) = true then
 let handshake_messages_hash = H((client_hello,client_key_share,
  server_hello,server_key_share,server_certificate)) in
 if verify(server_certificate_verify,handshake_messages_hash,pkS) =
  true then
 if server_finished = PRF(pre_master_secret,SERVER_FINISHED,
  H((client_hello,client_key_share,server_hello,server_key_share,
   server_certificate,server_certificate_verify))) then
(* master_secret *)
 let master_secret = PRF(hs_master_secret,EXTENDED_MASTER_SECRET,
  H((client_hello,client_key_share,server_hello,server_key_share,
   server_certificate,server_certificate_verify))) in
(* Finished *)
 let client_finished = PRF(pre_master_secret,CLIENT_FINISHED,
  H((client_hello,client_key_share,server_hello,server_key_share,
   server_certificate,server_certificate_verify,server_finished))) in
 let enc_client_finished = enc(client_finished,hs_master_secret) in
(* Client Output *)
 out(c, enc_client_finished);
(* event end - authentication *)
 event endClient(client_random,server_random,client_cipher_suites,
  master_secret);
(* key_block *)
 let key_block = PRF(master_secret,KEY_EXPANSION,
  (server_random, client_random)) in
 let (client_write_key:bitstring,server_write_key:bitstring,
  client_write_IV:bitstring,server_write_IV:bitstring) =
   (key_block_pt_client_write_key(key_block),
     key_block_pt_server_write_key(key_block),
      key_block_pt_client_write_IV(key_block),
       key_block_pt_server_write_IV(key_block)) in
(* additional_data, record_nonce *)
 let client_additional_data = (client_seq_num,client_type,
  client_record_version) in
 let client_record_nonce = xor(client_write_IV,client_seq_num) in
(* Application Data *) (* secrecy *)
 out(c, (client_record_nonce,client_additional_data,
  AEAD_Encrypt(client_write_key,client_record_nonce,AppDataClient,
   client_additional_data))).
```

上記の 1〜82 行目は，full ハンドシェイクにおけるクライアント処理の形式化である．

```
(* Server process *)
 let processServer(skS:bitstring,pkS:bitstring,
  server_certificate:bitstring,pkCA:bitstring) =
(* setup *)
 in(c, (server_version:bitstring,server_session_id:bitstring,
  server_cipher_suites:bitstring,server_seq_num:bitstring,
   server_type:bitstring,server_record_version:bitstring));
(* Server Intput *)
 in(c, (client_hello:bitstring,client_key_share:bitstring));
 let (client_version:bitstring,client_random:bitstring,
  client_session_id:bitstring,client_cipher_suites:bitstring,
   compression_methods:bitstring) = client_hello in
 if client_version = server_version &&
  client_cipher_suites = server_cipher_suites then
(* ServerHello *)
 new server_random:bitstring;
 let server_hello = (server_version,server_random,
  server_session_id,server_cipher_suites) in
(* ServerKeyShare *)
 new Y:bitstring;
 let server_key_share = exp(g, Y) in
(* pre_master_secret *)
 let pre_master_secret = exp(client_key_share,Y) in
(* session_hash *)
 let session_hash = H((client_hello,client_key_share,
  server_hello,server_key_share)) in
(* hs_master_secret *)
 let hs_master_secret = PRF(pre_master_secret,
  HANDSHAKE_MASTER_SECRET,session_hash) in
(* Certificate *)
 let enc_server_certificate =
  enc(server_certificate,hs_master_secret) in
(* CertificateVerify *)
 let handshake_messages_hash = H((client_hello,client_key_share,
  server_hello,server_key_share,server_certificate)) in
 let server_certificate_verify =
  sign(handshake_messages_hash,skS) in
 let enc_server_certificate_verify =
  enc(server_certificate_verify,hs_master_secret) in
(* Finished *)
 let server_finished = PRF(pre_master_secret,SERVER_FINISHED,
  H((client_hello,client_key_share,server_hello,server_key_share,
   server_certificate,server_certificate_verify))) in
 let enc_server_finished = enc(server_finished,hs_master_secret) in
(* master_secret *)
 let master_secret = PRF(hs_master_secret,EXTENDED_MASTER_SECRET,
  H((client_hello,client_key_share, server_hello,server_key_share,
   server_certificate,server_certificate_verify))) in
(* event begin - authentication *)
 event beginClient(client_random,server_random,
  server_cipher_suites,master_secret);
(* Server Output *)
 out(c, (server_hello,server_key_share,enc_server_certificate,
  enc_server_certificate_verify,enc_server_finished));
(* Server Intput *)
 in(c, enc_client_finished: bitstring);
 let client_finished = dec(enc_client_finished,hs_master_secret) in
(* check *)
 if client_finished = PRF(pre_master_secret,CLIENT_FINISHED,
  H((client_hello,client_key_share,server_hello,server_key_share,
   server_certificate,server_certificate_verify,server_finished)))
    then
(* key_block *)
 let key_block = PRF(master_secret,KEY_EXPANSION,
  (server_random,client_random)) in
 let (client_write_key:bitstring,server_write_key:bitstring,
  client_write_IV:bitstring,server_write_IV:bitstring) =
   (key_block_pt_client_write_key(key_block),
    key_block_pt_server_write_key(key_block),
     key_block_pt_client_write_IV(key_block),
      key_block_pt_server_write_IV(key_block)) in
(* additional_data, record_nonce *)
 let server_additional_data = (server_seq_num,server_type,
  server_record_version) in
 let server_record_nonce = xor(server_write_IV,server_seq_num) in
(* Application Data *)
 new AppDataServer:bitstring;
  out(c, (server_record_nonce,server_additional_data,
   AEAD_Encrypt(server_write_key,server_record_nonce,AppDataServer,
    server_additional_data))).
```

上記の 1〜80 行目は，full ハンドシェイクにおけるサーバ処理の形式化である．サーバにおける送受信は，クライアントにおける送受信関係と対になるように記述する．

```
(* Certificate Authority (CA) process *)
 let processCA(skCA:bitstring) =
  in(c, (HostInfoX:bitstring,pkX:bitstring));
 if HostInfoX <> HostInfoC && HostInfoX <> HostInfoS &&
  HostInfoX <> HostInfoCA then
 let signCA_pkX = sign(H((HostInfoCA,HostInfoX,pkX)),skCA) in
 let X_certificate = (HostInfoCA,HostInfoX,pkX,signCA_pkX) in
 out(c, X_certificate).

(* Main process *)
 process
(* Certificate Authority (CA): skCA,pkCA *)
 new skCA: bitstring;
 let pkCA = pk(skCA) in
 out(c, pkCA);
(* Server key: skS,pkS *)
 new skS: bitstring;
 let pkS = pk(skS) in
(* server_certificate *)
 let signCA_pkS = sign(H((HostInfoCA,HostInfoS,pkS)),skCA) in
 let server_certificate = (HostInfoCA,HostInfoS,pkS,signCA_pkS) in
 out(c, server_certificate);
 ((!processClient(pkCA)) |
  (!processServer(skS,pkS,server_certificate,pkCA)) |
   (!processCA(skCA)))
```

上記の 1〜8 行目は, full ハンドシェイクにおける認証局処理の形式化である．認証局の形式化においては，入力されたホストの情報及びそのホストの公開鍵よりそのホストの証明書を生成する処理を形式化している．また，上記の 11〜25 行目はメインプロセス処理の形式化である．メインプロセスでは，各プロセスの並列実行や秘密鍵，公開鍵，及び証明書の生成といった事前処理を記述する．

5 検証結果

TLS1.3 draft-06 における full ハンドシェイクを ProVerif を用いて形式化し，安全性検証を行った．秘匿についてはクライアントと攻撃者の間で鍵共有を成立させることができるかの検証を行った (アプリケーションデータの秘匿性)．認証については攻撃者がサーバになりすますことができるかの検証を行った．結果として，秘匿及び認証とも安全 (true) と出力された．すなわち，draft-06 における full ハンドシェイクに対する攻撃は発見されなかった．

6 まとめ

本稿では，ドラフトが頻繁に更新されている TLS1.3 の安全性評価の一環として，draft-06 における full ハンドシェイクについて ProVerif を用いて形式的に記述し，安全性検証を行った．検証の結果，攻撃は発見されなかった．今後の課題としては，draft-06 のみならず最新の draft における full ハンドシェイクの形式化及び full ハンドシェイク以外のプロトコルも形式化し，安全性検証を行う．さらに，AEAD の形式化についてはいくつかの方法が考えられるため，AEAD の形式化についての検討も行う．

参考文献

[1] B.Blanchet(Project leader) “ProVerif: Cryptographic protocol verifier in the formal model,” 2015. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[2] 独立行政法人情報処理推進機構セキュリティセンター, “SSL/TLS 暗号設定ガイドライン〜安全なウェブサイトのために (暗号設定対策編)〜 Ver.1.1,” 2015. http://www.ipa.go.jp/security/vuln/ssl_crypt_config.html.

[3] A.Freier, P.Karlton,and P.Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0,” RFC 6101, 2011.

[4] T.Dierks and C.Allen, “The TLS Protocol Version 1.0,” RFC 2246, 1999.

[5] T.Dierks and E.Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.1,” RFC 4346, 2006.

[6] S.Vaudenay, “Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS...,” Advances in Cryptology – EUROCRYPT 2002 Lecture Notes in Computer Science Volume 2332, pp.534–545, 2002.

[7] A.K.L.Yau, K.G.Paterson, and C.J.Mitchell, “Padding oracle attacks on CBC-mode encryption with secret and random IVs,” In Fast Software Encryption, pages 299–319, 2005.

[8] T.Dierks and E.Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246, 2008.

[9] M.Dworkin, “Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality,” NIST Special Publication 800-38C, 2004.

[10] M.Dworkin, “Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC,” NIST Special Publication 800-38D, 2007.

[11] P.Rogaway, “Authenticated-encryption with associated-data,” ACM Conference on Computer and Communications Security (CCS'02), pp.98–107, ACM press, 2002. http://web.cs.ucdavis.edu/~rogaway/papers/ad.html.

[12] E.Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft (work in progress),” 2015. https://tlswg.github.io/tls13-spec/.

[13] T.Jager, J.Schwenk, and J.Somorovsky, “On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption,” To appear in the 22nd ACM Conference on Computer and Communications Security (CCS), 2015. http://euklid.org/pdf/CCS15.pdf.

[14] E.Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft 06 (work in progress),” June, 2015. https://tools.ietf.org/html/draft-ietf-tls-tls13-06.

[15] D.Dolev and A.C.Yao, “On the Security of Public Key Protocols,” IEEE Transactions on Information Theory, IT-29(12), pp.198–208, 1983.

[16] M.Abadi and C.Fournet, “Mobile Values, New names, and Secure Communication,” Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01), pp.104–115, 2001.

[17] Bruno Blanchet, “Using Horn Clauses for Analyzing Security Protocols,” Formal Models and Techniques for Analyzing Security Protocols, volume 5 of Cryptology and Information Security Series, pp.86–111, 2011.

[18] B.Blanchet, “An Efficient Cryptographic Protocol Verifer Based on Prolog Rules,” In 14th IEEE Computer Security Foundations Workshop, pp.82–96, 2001.

[19] B.Blanchet, “From Secrecy to Authenticity in Security Protocols,” 9th International Static Analysis Symposium (SAS'02), volume 2477 of Lecture Note on Computer Science, pp.342–359, 2002.

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