なりすまし攻撃に対する Threemaハンドシェイクプロトコルの形式検証

中川 大輔[1,a)]　田中 篤[2]　木村 隼人[2,3]　山下 恭佑[2,4]　伊藤 竜馬[2,3]　五十部 孝典[2]

概要：Threema は，スイス，ドイツ，オーストリア，カナダ，オーストラリアを中心に世界で 1200 万人以上のユーザーおよび 8000 以上の組織に利用されている，暗号化メッセージングアプリケーションである．近年，Paterson ら (USENIX Security 2023) により，Threema のクライアント・サーバー間プロトコルにおけるハンドシェイクサブプロトコルに関して，クライアントやサーバーになりすますことが可能な攻撃がヒューリスティックに発見された．本研究では，形式検証ツール Tamarin を用いて，このハンドシェイクサブプロトコルを詳細にモデリングし，Paterson らの攻撃が自動的に検出可能であることを示す．さらに，攻撃に対する修正案についても同様の手法で再検証を行う．

キーワード：Tamarin Prover，形式検証，なりすまし攻撃

Formal Verification of the Threema Handshake Protocol against Impersonation Attacks

Daisuke Nakagawa[1,a)]　Atsushi Tanaka[2]　Hayato Kimura[2,3]　Kyosuke Yamashita[2,4]
Ryoma Ito[2,3]　Takanori Isobe[2]

Abstract: Threema is an encrypted messaging application used by more than 12 million users and over 8,000 organizations worldwide, primarily in Switzerland, Germany, Austria, Canada, and Australia. Recently, Paterson et al. heuristically discovered attacks on the handshake subprotocol of Threema's client-to-server protocol that enable an adversary to impersonate the client or the server(USENIX Security 2023). In this study, we model this handshake subprotocol in detail using the Tamarin-Prover and show that Paterson et al.'s attacks can be automatically detected. We further use the same method to verify a patched version of the protocol that mitigates these attacks.

Keywords: Tamarin Prover，Formal Verification，Impersonation Attack

1. はじめに

Threema はスイス発の暗号化メッセージングアプリケーションであり，世界で 1200 万人以上のユーザーと 8000 以上の組織に利用されている [1]．スイス，ドイツ，オーストリア，カナダ，オーストラリアでは，有料カテゴリの Android アプリにおいて上位に位置する．Threema はスイス政府およびスイス軍に採用されており，全ての公式通信での使用が義務付けられている．Threema 社は，高いセキュリティや，米国の管轄外であるスイス製である点を長所としている．

E2EE（エンドツーエンド暗号化）は送信者と受信者間の通信路上の暗号化で，第三者による盗聴や改竄を防ぐものである．近年のセキュアメッセージングプロトコルは，E2EE の通信を提供するために設計されている．国家によ

[1] 兵庫県立大学
University of Hyogo
[2] 大阪大学
The University of Osaka
[3] NICT (情報通信研究機構)
[4] AIST (産業技術総合研究所)
[a)] ad25f049@guh.u-hyogo.ac.jp

る個人情報の収集が明らかになったことを受け，多くのプロトコルにおいて，機密性，完全性，認証だけでなく，前方秘匿性および後方秘匿性が求められている [2]．前方秘匿性は，セッション鍵や長期秘密鍵が漏洩しても，過去に完了した通信が保護される性質であり，後方秘匿性は，過去や現在の鍵情報が漏洩しても，将来の通信が安全であることを保証する性質である．これらの性質は，鍵漏洩時の被害を時間的に限定し，継続的な安全性を確保するために不可欠である．

Paterson ら [3] は，Threema のプロトコル群に着目し，ヒューリスティックな解析により，複数の攻撃を報告した．その一つであるなりすまし攻撃は，C2S（クライアントツーサーバー）プロトコルのハンドシェイクにおいてクライアント側の一時秘密鍵が一つ漏洩するだけで，該当セッションを超えて持続的ななりすましが成立し得るという深刻な性質を明らかにしたものである．この攻撃は，一時秘密鍵とクライアント認証用の `vouch` を再利用することで，攻撃者がサーバーに対して，被害者を継続的に装うことを可能とする．結果として，C2S プロトコルが提供していた前方秘匿性を破壊し得るもので，E2E 暗号化層にも影響を及ぼした．

形式検証ツールである Tamarin Prover [4] は，Dolev-Yao モデルに基づく記号的モデリングを用い，並列実行や鍵漏洩を含む多様な実行トレース上で，機密性や完全性，認証などの安全性を自動検証可能である．形式検証ツールを用いるメリットは，設計上の脆弱性を手動解析では見落とし得る場合でも，反例トレースとして提示できる点である．近年では TLS1.3 [5] や 5G-AKA [6] をはじめとする実運用プロトコル検証にも適用されている．しかし，Tamarin Prover は記述されたルールに基づき全ての実行トレースを探索するため，安全性検証に膨大な時間を要する場合がある．したがって，検証の計算量を削減することが重要な課題となる．

本研究では，Tamarin Prover を用いて，C2S プロトコルのハンドシェイクサブプロトコルに対して，Paterson らのなりすまし攻撃が検出可能かを形式的に評価することを目的とする．手動評価ではなく自動評価可能とすることで，一時秘密鍵漏洩時の評価手法を一般化し，複数の暗号プロトコルに対する同様の評価することが可能となる．実際，彼らの論文を元に修正されたプロトコルが同様の攻撃に耐性を持つか検証することを試みる．具体的には状態遷移をルールとして記述することでプロトコルをモデリングし，それに対して Pateson らのなりすまし攻撃を検証する `lemma` を記述し，検証した．その上で探索空間の爆発を防ぐために，ユーザーを固定する制約や，今回の検証において非本質的なナンスの除去を行った上で検証を実施する．

結果としてなりすまし攻撃は，Tamarin により反例トレースを可視化でき，検出可能であった．一方で，修正後のプロトコルについては，記述した `lemma` の自動証明が収束せず，安全性の証明には至らなかった．この課題に対処するため，本研究では計算量増大の原因について考察し，解決案を示す．

本論文の構成は以下のとおりである．2 章では Threema プロトコルの概要を示し，その中でも特に C2S プロトコルにおけるハンドシェイクサブプロトコル，Paterson らのなりすまし攻撃，および修正後のプロトコルについて述べる．さらに，攻撃者モデルと Tamarin の概要について述べる．3 章では修正前のプロトコルに対する Tamarin を用いたなりすまし検証について述べ，4 章では修正後のプロトコルに対する検証を記述する．5 章では，本研究における検証結果の考察と今後の課題について述べ，6 章で全体を総括する．

2. 前提知識

2.1 Threema プロトコル

Threema のプロトコルは標準的な暗号プリミティブに基づいており，主に使用される暗号ライブラリは Networking and Cryptography Library (NaCl) [7] である．これは Curve25519 Diffie-Hellman 鍵交換と AEAD アルゴリズムである XSalsa20-Poly1305 による暗号化で構成される．

長期鍵は，秘密鍵を小文字，公開鍵を大文字で表記する．例えば，ユーザー $\mathcal{A}$ についての 32 バイト秘密鍵と 32 バイト公開鍵のペアを (a, A) と記述する．生成元 g が与えられたとき，公開鍵 A は $A = g^a$ により生成される．$X25519$ は秘密鍵と公開鍵を入力とし，楕円曲線上のスカラー倍算によって共有秘密を出力するものである．

ユーザーが新規アカウントを作成して Threema サーバーに登録する際には登録プロトコルが実行される．これによりサーバーはユーザーに対して新しい Threema ID を発行する．その後，ID とユーザーの長期公開鍵をデータベースに保存する．

Threema は，E2E プロトコルと C2S プロトコルという 2 つの暗号プロトコルで構成されている．E2E プロトコルは，長期鍵を利用してクライアント間のメッセージ内容を第三者が取得できないように暗号化し，C2S プロトコルはクライアントからサーバーへの転送の際にこれらのメッセージを保護する役割を担う．

C2S プロトコルはさらに，ハンドシェイクサブプロトコルとトランスポートサブプロトコルという 2 つのサブプロトコルで構成されている．ハンドシェイクサブプロトコルはクライアントとサーバーが通信してセッション鍵を確立し，トランスポートサブプロトコルは確立されたセッション鍵を使用してクライアントとサーバーがメッセージを交換する．C2S プロトコルでは，各ユーザーは一つのデバイスからのみサーバーに接続することができる．同一 ID で同時に複数のデバイスから接続した場合，新しい接続が優

先され，既存の接続は切断される．その際，切断されたクライアントは，別のデバイスが接続したことが示す通知を受け取る．

2.1.1 C2S のハンドシェイク

C2S プロトコルのハンドシェイクサブプロトコルについて詳しく解説する．ここでの C2S プロトコルは，Paterson らの攻撃 [3] が提案された当時の仕様に基づき，本研究ではこれを修正前のプロトコルと呼ぶ．

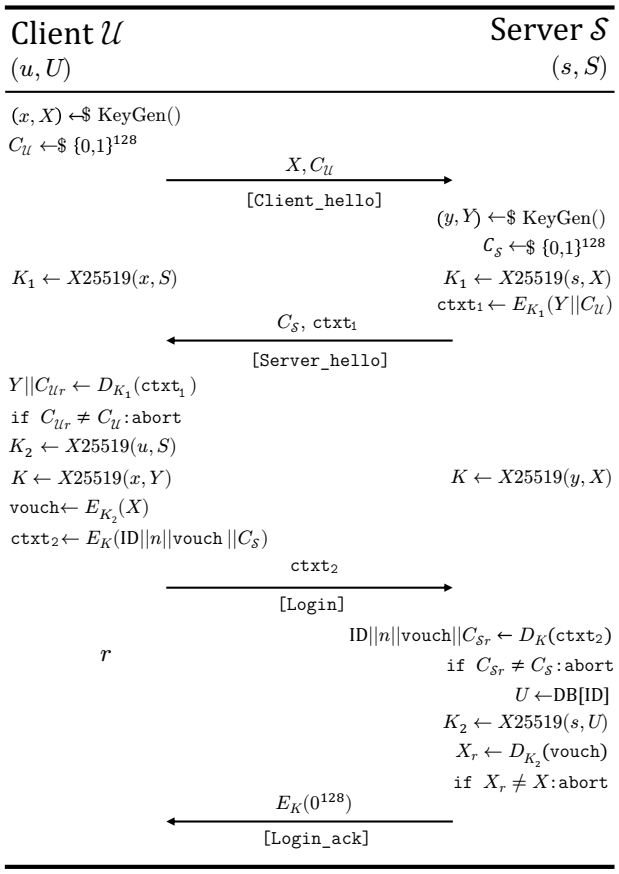

図 1 Threema の C2S プロトコル（ハンドシェイクサブプロトコル）．簡潔化のため，vouch 生成に用いられるナンス以外は省略．DB は Threema ID を公開鍵に対応付けるデータベースを抽象化したものである．

C2S のハンドシェイクサブプロトコルはサーバー $\mathcal{S}$ とクライアント $\mathcal{U}$ の間で定義される．プロトコルの状態遷移を図 1 に示す．サーバー $\mathcal{S}$ の長期鍵ペアを $(s, S = g^s)$，クライアント $\mathcal{U}$ の長期鍵ペアを $(u, U = g^u)$ とする．クライアント側ではサーバーの長期公開鍵がアプリに事前に埋め込まれており，サーバー側は登録プロトコルでクライアントの長期公開鍵を受け取っているため，両者は互いの長期公開鍵を知っていると仮定できる．

クライアントは 32 バイトの一時的な鍵ペア $(x, X = g^x)$ と 16 バイトの値であるクライアントクッキー $C_{\mathcal{U}}$ を生成し，$(X, C_{\mathcal{U}})$ をサーバーに送信する（Client_hello）．

サーバーはこれらを受信すると，32 バイトの一時的な鍵ペア $(y, Y = g^y)$ と 16 バイトの値であるサーバークッキー $C_{\mathcal{S}}$ を生成する．次に，サーバーは長期秘密鍵 s とクライアントの一時公開鍵 X から $K_1 = X25519(s, X)$ を計算し，その鍵を用いて，$(Y, C_{\mathcal{U}})$ を暗号化する．サーバーは暗号文と $C_{\mathcal{S}}$ をクライアントに送信する（Server_hello）．

クライアントは受信後，同様に K_1 を計算し，暗号文を復号する．また，得られたクライアントクッキーが自身が生成した $C_{\mathcal{U}}$ と一致することを確認する．復号が失敗する場合，またはクッキーが一致しない場合はクライアントはプロトコルを中止する．クライアントはさらに，長期鍵同士の共有鍵 $K_2 = (u, S)$ と一時鍵同士の共有鍵 $K = (x, Y)$ を計算する．クライアントは X を K_2 で暗号化したものを vouch とし，クライアント ID，vouch 用ナンス，vouch，および $C_{\mathcal{S}}$ をまとめてを K で暗号化し，サーバーに送信する（Login）．

サーバーは受信後，同様に K と K_2 を計算する．まず K で復号し，次に得られた vouch を K_2 で復号する．復号が失敗する場合，復号により得られたサーバークッキーが自身が生成した $C_{\mathcal{S}}$ と一致しない場合，または vouch の復号結果がクライアントの送信した X と一致しない場合，サーバーはプロトコルを中止する．認証が成功した場合，サーバーは 0^{128} を K で暗号化してクライアントに送信する（Login_ack）．

クライアントとサーバーのクッキーは，リプレイ攻撃の防止と，トランスポートサブプロトコルにおけるナンス生成用カウンターの初期化に使用される．また，鍵 K は，トランスポートサブプロトコルにおけるセッション鍵として用いられる．

2.1.2 修正前のプロトコルに対するなりすまし攻撃

Paterson らのクライアントなりすまし攻撃について解説する．前提条件として，攻撃者がクライアントの C2S ハンドシェイクで使用した一時秘密鍵 x のいずれかを入手していると仮定する．この一時秘密鍵漏洩は，乱数生成器の不具合やサイドチャネル攻撃により発生する可能性がある．さらに，攻撃者は対応する C2S セッションを受動的に記録しているとする．

この条件下で，攻撃者 $\mathcal{E}$ はクライアントになりすますことが可能である．漏洩セッションとなりすましセッションの遷移を図 2 に示す．攻撃者は x を知っているため，サーバーの長期公開鍵 S を用いて K_1 を計算することが可能である．これにより，サーバーからクライアントへの暗号文を復号し，サーバーの一時公開鍵 Y を取得する．さらに，セッション鍵 K を計算し，暗号文を復号することで vouch を取得することができる．攻撃者はこの x と vouch を再利用して，サーバーと通信することにより，長期鍵同士の共有鍵 K_2 を知らずにクライアントになりすますことが可能である．

vouch は繰り返し再利用可能であるため，攻撃者は永続

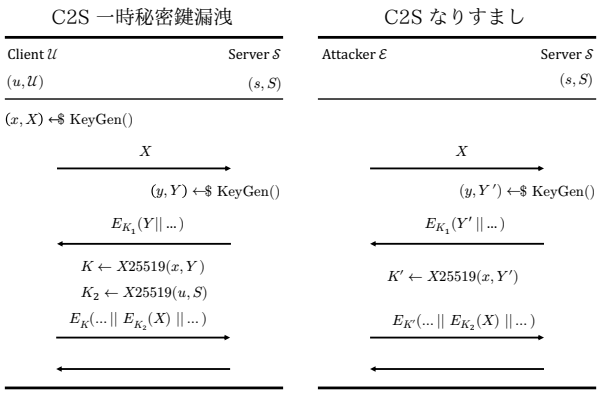

図 2 クライアントなりすまし攻撃．攻撃者は一時秘密鍵漏洩セッションを盗聴し，なりすましセッションにおいて，漏洩した x に対する一時公開鍵 X と vouch をリプレイすることによってなりすます．特に記述がない部分は図 1 と同様である．

的に被害者になりすましてサーバーに接続し，被害者あての E2E 暗号化メッセージにアクセスできるようになる．これにより，E2E プロトコルが存在するにも関わらず，攻撃者は被害者が誰といつ通信しているかを把握可能となる．この攻撃は Threema における C2S プロトコルが提供する前方秘匿性を破壊し得るものである．攻撃者と被害者が同時にサーバーに接続しない限り，被害者は攻撃を検知不可能である．もし，同時接続しても，攻撃者はサーバーからの通知を監視し，接続を終了することが可能である．

さらに，当時の C2S プロトコルには，アプリが再起動されない限り，クライアントは最大で 7 日間同じ一時鍵を再利用し続ける仕様が存在した．このため，その期間中に x が漏洩すると，攻撃者は K および K_1 を計算することができ，サーバーになりすますことも可能となる．結果として，攻撃者は被害者の送信する Threema サーバー宛の E2E 暗号化メッセージも読み取ることができ，E2E プロトコルに対する攻撃も可能となる．同一一時鍵を長期間再利用することにより，この攻撃の影響は大幅に拡大する．

2.1.3 修正後のプロトコル

クライアントへのなりすまし攻撃に対する対策の一つとして，vouch にクライアントの一時公開鍵 X に加えてサーバークッキー C_S を含めるよう修正が行われた．さらに，サーバーへのなりすまし攻撃も防ぐため，それぞれ一回のセッションごとに一時秘密鍵を更新するよう仕様が変更された．そのため，本研究ではこれらの修正が行われたプロトコルを修正後のプロトコルと呼ぶ．

2.2 攻撃者モデル

この C2S のハンドシェイクに対する攻撃者モデルは，アプリを実行しているデバイスと Threema サーバー間の通信を監視可能な攻撃者である．このモデルは，安全なネットワークプロトコルにおける標準的な攻撃者モデルに相当し，Dolev-Yao 攻撃者とも呼ばれる．

ただし，後述の 3 章における検証では，モデリング上の制約により，攻撃者には二つの制限を課している．これらの制限は，クライアントとサーバー間の通信をその当事者間でのみ利用できること，および AEAD 暗号化におけるナンスや関連データを改竄できないことである．

2.3 Tamarin Prover

2.3.1 概要

Tamarin Prover[4] は，セキュリティプロトコルの記号的モデリングおよび形式的検証を行う自動検証ツールであり，無限に多くのプロトコルセッションや並列実行，および強力な攻撃者モデルを考慮した解析が可能である．このツールは暗号プロトコル解析には広く用いられる Dolev-Yao モデルを基盤とし，暗号プリミティブは暗号アルゴリズムの内部構造を考慮せず理想化され，いずれもブラックボックスとして扱われる．

Tamarin は，プロトコルの仕様，攻撃者モデル，および検証すべきセキュリティ性質を入力として，全ての可能な実行トレースに対し，特性の成立を検証する．Tamarin からは，性質を満たす場合は `verified`，満たさない場合は `falsified` が出力される．しかしながら Tamarin は全探索を行うため，与えられた検証が必ずしも停止するとは限らず，証明が完了しない場合もある．

2.3.2 モデリングと検証

プロトコルと攻撃者の振る舞いはマルチセット書き換えルールによって記述される．例えば以下のように表される．

```
1 rule Example:
2 [Pre(x)]--[Action(x)]->[Post(x)]
```

ルール内の要素 `Pre(x)`，`Action(x)`，`Post(x)` はそれぞれファクトといい，状態を表すものである．左辺 `Pre(x)` は適用前の前提（ファクトの多重集合），右辺 `Post(x)` は適用後に生成されるファクト，中央の `Action(x)` はルール適用時に実行トレースへ記録されるアクションファクトである．ルールは，現在の状態において `Pre(x)` と単一化できるファクトが存在すれば適用可能であり，そのファクトを消費して，`Post(x)` に遷移する．アクションファクト `Action(x)` の記述は必須ではないが，`lemma` によって安全性特性を定義する際の参照用ラベルとして用いられる．なお，`Action(x)` や `Post(x)` に記述される変数は原則として全て `Pre(x)` に記述されていなければならない（`$`で始まる公開変数のみ例外）．これにより遷移系の意味論が良定義となり，モデルの一貫性が保証される．

安全性特性は，アクションファクトと，時間インデックスを用いた一階述語論理の `lemma` として記述される．これにより，機密性，完全性，認証性など多様な安全性特性を統一的に表現・検証可能である．Tamarin には intaractive

モードが存在し，標準搭載されている GUI を利用することで各変数の状態遷移を可視化し，デバッグすることが可能である．

3. 修正前のプロトコルに対するなりすまし攻撃の検証

本章では修正前のプロトコルに対するなりすまし攻撃を Tamarin で検証するためのモデリングについて述べる．

図 1 に示す全体構造を忠実にモデリングする場合，探索空間が膨大となる問題が発生する．AEAD を再現する際に，ナンスを含めてモデリングを行ったが，一つのモデリング条件を満たすトレースを探索する exist-trace においても，メモリ不足により検証が不可能であった．これは，Tamarin 内の変数の状態遷移が無数に考慮され，探索空間が爆発したことが原因と考えられる．このことから，ネットワーク上で扱う情報量が増加するにより，攻撃者の選択肢が増大し，結果として検証が困難になることが確認された．

3.1 ルール・制約

修正前のプロトコルのモデリングは基本的に，図 1 におけるクライアントとサーバーがそれぞれひとまとまりで実行する部分をルールの一つの単位として記述する．探索空間を削減し，かつなりすまし攻撃を効率的に検出するため，本研究ではモデリングに対して複数の制約を付与する．これにより無関係な遷移を排除し，検証効率の向上を図る．

以下がモデリングのルールと制約である．

```
builtins:
diffie-hellman, symmetric-encryption

rule LongkeyC:
    [ Fr(~ltkC) ]
     -->[ !InitClient($ID, ~ltkC) ]

rule LongkeyS:
    [ Fr(~ltkS) ]
    -->[ !InitServer($S, ~ltkS) ]

rule InitClient:
    let
      pkC = 'g'^~ltkC
    in
    [ !InitClient($ID, ~ltkC) ]
    --[ InitC( )]->
    [ !LtkC( $ID, ~ltkC), !PkC( $ID, pkC),
     Out(pkC) ]

rule InitServer:
    let
      pkS = 'g'^ ~ltkS
    in
    [ !InitServer($S, ~ltkS) ]
    --[ InitS( ) ]->
    [ !LtkS( $S, ~ltkS), !PkS( $S, pkS)
    , Out(pkS) ]

rule Client_hello:
    let
      epkC = 'g'^~ekC
    in
    [ Fr(~ekC), Fr(~Cu),
    !LtkC( $ID, ~ltkC) ]
    -->[ EskC( $ID, ~ekC)
    , EpkC( $ID, epkC)
    , Client_hello_after( ~Cu, $ID, ~ltkC,
     ~ekC, epkC ), Out(< epkC, ~Cu>) ]

rule Server_hello:
    let
      key1 = epkC^~ltkS
      epkS = 'g'^~ekS
      ctxt1 = senc( pair(epkS, ~Cu), key1)
    in
    [ In(< epkC, ~Cu>), !LtkS( $S, ~ltkS)
    , Fr(~ekS), Fr(~Cs) ]
    -->[ EskS( $S, ~ekS )
    , EpkS( $S, epkS )
    , KeyoneS( ~Cs, key1 )
    , Server_hello_after( ~Cs, $S, ~ltkS,
    epkC, ~ekS ), Out(<~Cs, ctxt1>) ]

rule Client_Server_hello:
    let
      key1 = pkS^~ekC
    in
    [ Client_hello_after( ~Cu, $ID, ~ltkC,
     ~ekC, epkC ), !PkS( $S, pkS) ]
    -->[ Client_login_before( ~Cu, $ID, $S
    , key1, ~ltkC, ~ekC, epkC, pkS ) ]

rule Login:
    let
      epkS = fst(sdec(ctxt1, key1 ))
      Cur = snd(sdec(ctxt1, key1 ))
      key2 = pkS^~ltkC
      ssessionkey = epkS^~ekC
      vouch = senc( epkC, key2 )
      ctxt2 = senc(pair($ID, pair(vouch, ~
    Cs)), ssessionkey)
    in
    [ Client_login_before( ~Cu, $ID, $S,
    key1, ~ltkC, ~ekC, epkC, pkS )
    , In(<~Cs, ctxt1>)]
    --[ Client_l( $ID, $S, epkC )
    , Eq( ~Cu, Cur)]->
    [ Client_login_after( ~Cu, ~Cs, $ID,
```

```
    $S, ssessionkey, ~ekC, epkC), Out(ctxt2
    ) ]
73
74 rule Server_Client_login:
75     let
76       ssessionkey = epkC^~ekS
77     in
78     [ Server_hello_after( ~Cs, $S, ~ltkS,
    epkC, ~ekS ) ]
79     -->[ Server_login_ack_before( ~Cs, $S,
        ssessionkey, ~ltkS, epkC, ~ekS )]
80
81 rule Login_ack:
82     let
83       text2 = sdec( ctxt2, ssessionkey )
84       vouchr = fst(snd(text2))
85       Csr = snd(snd(text2))
86       CIDr = fst(text2)
87       key2 = pkC^~ltkS
88       epkCr = sdec( vouchr, key2 )
89     in
90     [ Server_login_ack_before( ~Cs, $S,
    ssessionkey, ~ltkS, epkC, ~ekS )
91     , !PkC( $ID, pkC), In(ctxt2) ]
92     --[ Server_l_a ( $ID, $S, epkC )
93     , Eq(~Cs, Csr), Eq(epkC, epkCr)]->
94     [ Out(senc('zeros_128bit', ssessionkey
    )) ]
95
96 rule Client_recieve_ack:
97     [ In(senc('zeros_128bit', ssessionkey)
    ), Client_login_after( ~Cu, ~Cs, $ID,
    $S, ssessionkey, ~ekC, epkC) ]
98     --[ Client_r_a($ID, $S, ~ekC, epkC)
    ]->
99     [ ]
100
101 restriction Equality:
102   "All a b #i. Eq(a,b)@i ==> a=b"
103
104 restriction long_key_server:
105   "All #i #j. InitS() @ i & InitS() @ j
      ==> #i = #j"
106
107 restriction long_key_client:
108   "All #i #j. InitC() @ i & InitC() @ j
      ==> #i = #j"
```

図 1 と対応するように，各ルールを記述している．鍵の変数は，長期秘密鍵を ltk，長期公開鍵を pk，一時秘密鍵を ek，一時公開鍵を epk として表す．さらに，その後ろに C か S のどちらが付くかにより，クライアントとサーバーを区別する．

Threema の C2S プロトコルでは AEAD が使用されているが，先に述べたように，そのまま純粋に AEAD を実装するだけでは探索空間が膨大になる．そのため，今回のなりすまし攻撃検証において，本質的には関係のないナンスや関連データの変数を削除し，symmetric-encryption（共通鍵暗号）を用いることで探索空間の削減を図る．これにより，攻撃者は AEAD のナンスや関連データを改竄することができない設定と等価になる．しかし，今回の目的はなりすましの検出であるため，この制約は大きなボトルネックにはなりえないと考えられる．また，restriction Equality は，アクション中に Eq(a,b) で記述された a と b が必ず一致することを強制する制約である．これは認証処理が必ず成功することを前提としたモデリングに相当する．Threema のハンドシェイクでは，復号に失敗した場合やリプレイ対策用のクッキーが一致しない場合には通信が中止される．Paterson らのなりすまし攻撃は，この認証処理を通過する攻撃であるため，本研究で用いる制約条件の下でも攻撃の検証は可能である．

また，restriction で記述している long_key_server と long_key_client は，クライアント側および，サーバー側で対応する特定のルールがそれぞれ一度しか発生しない制約である．この結果，本モデルにおいて，単一クライアントと単一サーバーに固定され，攻撃者は両者間の通信から得られた情報のみを用いて攻撃を行うことが可能となる．

一時秘密鍵漏洩と攻撃者が自身で乱数を生成するためのルールをそれぞれ以下に示す．

```
1 rule Reveal_ephemeral_key_client:
2     [ EskC( ID, ekC) ]
3     --[ Reveal_ekC(ID, ekC) ]->
4     [ Out( ekC ) ]
5
6 rule Attack_hello:
7     [ Fr(~Ca) ]
8     -->[ Out( ~Ca ) ]
```

Reveal_ephemeral_key_client は Paterson らのなりすまし攻撃を成立させるために必要な前提条件である．これを設定しない場合，攻撃者は通信路上で取得した情報のみを用いた書き換えしか行えない制約となってしまう．また，Attack_hello により，攻撃者はクッキーや鍵を自ら生成可能になる．

3.2 lemma

なりすまし攻撃を検証する lemma について記述する．

```
1 lemma client_impersonation_attack:
2 "(All ID S epkC #i.
3     (Server_l_a ( ID, S, epkC ) @ #i)
4     ==> (Ex #a. Client_l( ID, S, epkC ) @
      a
5     & (All #j. Server_l_a ( ID, S, epkC )
      @ #j ==> #i = #j)
6     )
```

```
 7 )"
 8
 9 lemma server_impersonation_attack:
10 "(All ID S ekC epkC #i.
11     Client_r_a( ID, S, ekC, epkC ) @i
12     ==> Ex #j. Server_l_a ( ID, S, epkC )
       @j
13 )"
```

client_impersonation_attack はクライアントなりすましを検証するための lemma である．この lemma は，同一の一時公開鍵に対して，サーバーがハンドシェイクを再実行行わないこと，すなわちリプレイ攻撃の有無を検証するものである．また，server_impersonation_attack はクライアント側でハンドシェイクが完了した際に，実際に通信していたサーバーが存在することを確認する，すなわちサーバーなりすましを検証するための lemma である．検証の結果，lemma はいずれも falsified となり，Paterson らの攻撃を検出した．

4. 修正後のプロトコルに対するなりすまし攻撃の検証

本章では修正後のプロトコルに対するなりすまし攻撃を Tamarin で検証するためのモデリングについて述べる．修正箇所は，vouch 部分にクライアントクッキー C_S を含めること，ならびに各セッションにおいて一時鍵を更新することである．

4.1 ルール・制約

ルールおよび制約は基本的に 3.1 節で示したものと同一である．変更点は，vouch 部分に関連するルールのみであり，X に加えて，サーバークッキー C_S が含まれる点が異なる．同じルールについてはソースコードを割愛し，変更点の Login のみ，以下に示す．

```
 1 rule Login:
 2     let
 3       epkS = fst(sdec(ctxt1, key1 ))
 4       Cur = snd(sdec(ctxt1, key1 ))
 5       key2 = pkS^~ltkC
 6       ssessionkey = epkS^~ekC
 7       vouch = senc( <epkC, ~Cs>, key2 )
 8       ctxt2 = senc(pair($ID, pair(vouch, ~
       Cs)), ssessionkey)
 9     in
10     [ Client_login_before( ~Cu, $ID, $S,
       key1, ~ltkC, ~ekC, epkC, pkS )
11     , In(<~Cs, ctxt1>)]
12     --[ Client_l( $ID, $S, epkC )
13     , Eq( ~Cu, Cur)]->
14     [ Client_login_after( ~Cu, ~Cs, $ID,
       $S, ssessionkey, ~ekC, epkC), Out(ctxt2
       ) ]
```

Login_ack で，サーバーはこれを復号し，サーバークッキーについても追加で認証を行う．

4.2 lemma

クライアントなりすましの検証には 3.2 節で示した client_impersonation_attack をそのまま適用する．これにより，クライアントの一時秘密鍵が漏洩した場合であっても，修正後のプロトコルが，同様のクライアントなりすましに対して安全かを検証可能となる．サーバーなりすましに関しては以下の lemma を用いて検証を行う．

```
1 lemma server_impersonation_attack_two:
2 "(All ID S ekC epkC #i.
3     Client_r_a(ID, S, ekC, epkC) @i
4     & not(Ex #r. Reveal_ekC( ID, ekC) @ r)
       )
5     ==> Ex #j. Server_l_a ( ID, S, epkC )
      @j
6 )"
```

修正後のプロトコルでは，一時鍵を使い回さずにセッション毎に交換する．そのため，そのセッション開始時ではクライアントの鍵が漏洩していないとみなすことができる．したがって，本検証は，クライアントの一時秘密鍵が漏洩していない場合に，サーバーなりすまし攻撃に対する安全性を検証するものである．結果として，検証した lemma はいずれも終了せず，証明が完了しなかった．この理由については次章で考察する．

5. 考察

Tamarin における検証では，サーバーの一時秘密鍵や両者の長期秘密鍵，共有鍵の漏洩ルールも併せて記述するのが一般的である．そのため，本研究でもこれらを記述し，ルールが発生しないという条件下での検証も実施した．全てにおいて同様の結果が得られたが，漏洩ルールを追加するほど，検証に要する時間が増加した．さらに，修正後のプロトコルについても同様に検証を行ったが，いずれも証明は完了しなかった．

修正後のプロトコルにおいて漏洩ルールの増加に伴い検証時間が増加すること，ならびに 4.2 節の lemma で検証が完了しないことから，修正後のプロトコルについても漏洩ルールを追加した検証は困難であることが考えられる．

今回の検証では，修正前のプロトコルに対しては，なりすまし攻撃を検出でき，falsified が出力されたため，攻撃が見つかった時点で検証が終了した．一方で，修正後のプロトコルに対しては，攻撃は検出できず，証明が完了しなかった．これより，モデリング時に変数を削除して探索空間を減らすアプローチでは依然として検証完了には至らないことが確認された．しかしながら，今回対象とした

C2S ハンドシェイクよりも複雑な TLS1.3 の tamarin 検証事例 [5] では sublemma を用いて複雑なプロトコルの安全性を検証している．また，ルールや分岐に優先度を与え，証明を順位付けすることにより，理論的に網羅性を維持しつつも効率的に探索することを可能とする Heuristics[8] も存在する．

今後はこれらの手法を適用し，探索方法を工夫することにより，探索時間の短縮およびメモリ超過の回避を図り，証明を完了できるか検証することが課題である．

6. まとめ

本研究では，修正前のプロトコルを対象に，形式検証ツール Tamarin Prover を用いてモデリングし，Paterson らのなりすまし攻撃が再現可能かを検証した．ユーザー数や，AEAD の部分に制約を課すことで，攻撃を検出し，反例トレースを検出することに成功した．一方，修正されたプロトコルについては探索が完了せず，証明に至らなかった．

今後の展望として，まずは 5 章で述べた sublemma と Heuristics を導入し，修正後のプロトコルに対するなりすまし攻撃に関する証明を完了できるか検証する．また，暗号プロトコルが満たすべき他の性質についても検証を行うとともに，一時秘密鍵漏洩時の検証手法を他の暗号プロトコルへ応用する予定である．

謝辞

本研究は，JSPS 科研費 JP24H00696 と JST AIP 加速課題 JPMJCR24U1 の支援を受けたものである．

参考文献

[1] Threema GmbH: Why - Threema (2025). `https://threema.com/en/why-threema` (Accessed: 2025-08-18).

[2] Unger, N., Dechand, S., Bonneau, J., Fahl, S., Perl, H., Goldberg, I. and Smith, M.: SoK: Secure Messaging, *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, pp. 232–249 (online), DOI: 10.1109/SP.2015.22 (2015).

[3] Paterson, K. G., Scarlata, M. and Truong, K. T.: Three Lessons From Threema: Analysis of a Secure Messenger, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023* (Calandrino, J. A. and Troncoso, C., eds.), USENIX Association, pp. 1289–1306 (2023). `https://www.usenix.org/conference/usenixsecurity23/presentation/paterson`.

[4] Meier, S., Schmidt, B., Cremers, C. and Basin, D. A.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings* (Sharygina, N. and Veith, H., eds.), Lecture Notes in Computer Science, Vol. 8044, Springer, pp. 696–701 (online), DOI: 10.1007/978-3-642-39799-8_48 (2013).

[5] Cremers, C., Horvat, M., Hoyland, J., Scott, S. and van der Merwe, T.: A Comprehensive Symbolic Analysis of TLS 1.3, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (Thuraisingham, B., Evans, D., Malkin, T. and Xu, D., eds.), ACM, pp. 1773–1788 (online), DOI: 10.1145/3133956.3134063 (2017).

[6] Basin, D. A., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R. and Stettler, V.: Formal Analysis of 5G Authentication, *CoRR*, Vol. abs/1806.10360 (online), available from ⟨http://arxiv.org/abs/1806.10360⟩ (2018).

[7] Bernstein, D. J., Lange, T. and Schwabe, P.: NaCl: Networking and Cryptography Library (2011). domain high-speed software library for cryptography.

[8] The Tamarin Team: *Tamarin Prover Manual: Advanced Features* (2024). `https://tamarin-prover.com/manual/master/book/011_advanced-features.html` (Accessed: 2025-08-18).