ProVerifによる トークン型電子現金プロトコルの形式検証

山本 輪（長崎大学），○江島 奨悟（長崎大学），
奥田 哲矢（NTT社会情報研究所），荒井 研一（長崎大学）

始めに

第99回CSEC研究会
”トークン型電子現金方式の形式検証手法に関する初期検討”[1]

形式検証ツールProVerifを用いた検証
・プロトコル形式化について
・形式化した際の工夫点について

検証対象となる電子現金
・トークン型
・間接発行方式
・利用者の間でオフライン取引可能
・点々流通可能(transferable)

[1] 奥田哲矢, 荒井研一, 齋藤恆和, 千田浩司, 中林美郷, 山村和輝, 宮澤俊之, 阿部正幸, “トークン型電子現金方式の形式検証手法に関する初期検討”, 情報処理学会研究報告書, Vol.2022-CSEC-99 No.24,2022.

安全性モデル

安全性要件

- 秘匿性　(Secrecy)
- 認証性　(Authentication)
- 偽造不可能性　(Unforgeability)
- 二重使用特定性　(Double Spend-Identification)
- 免責性　(Exclupability)

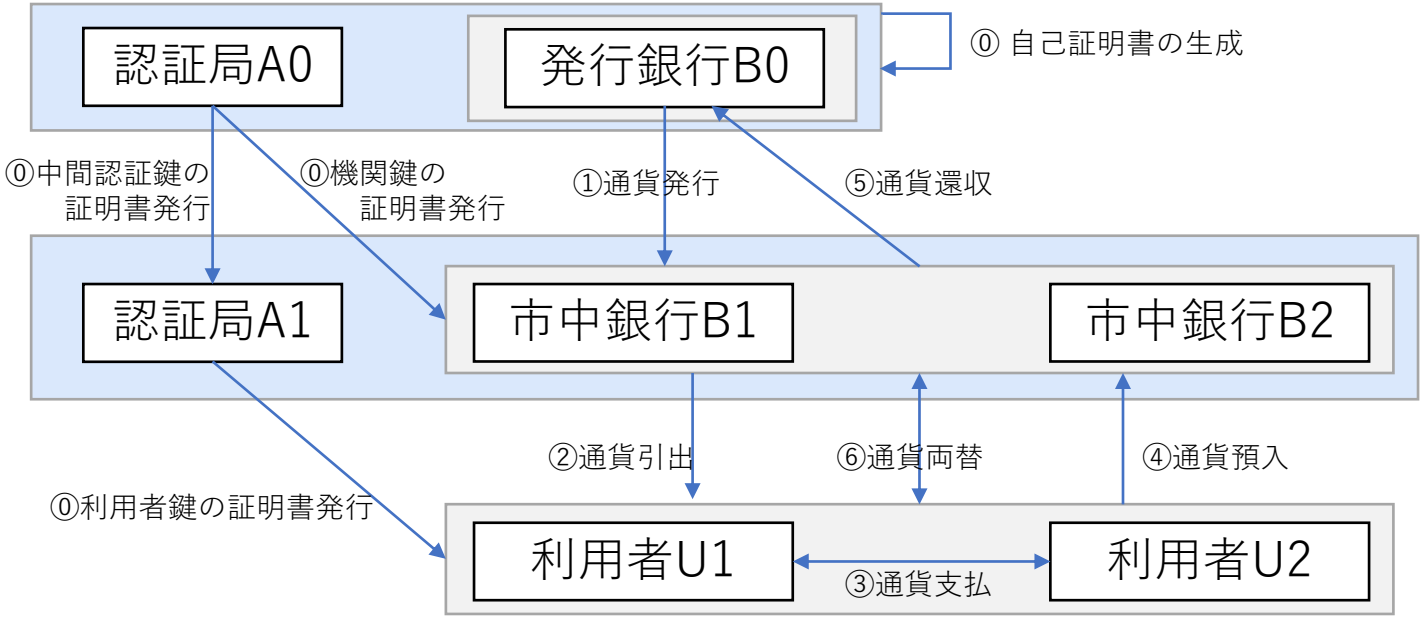
プライバシー要件

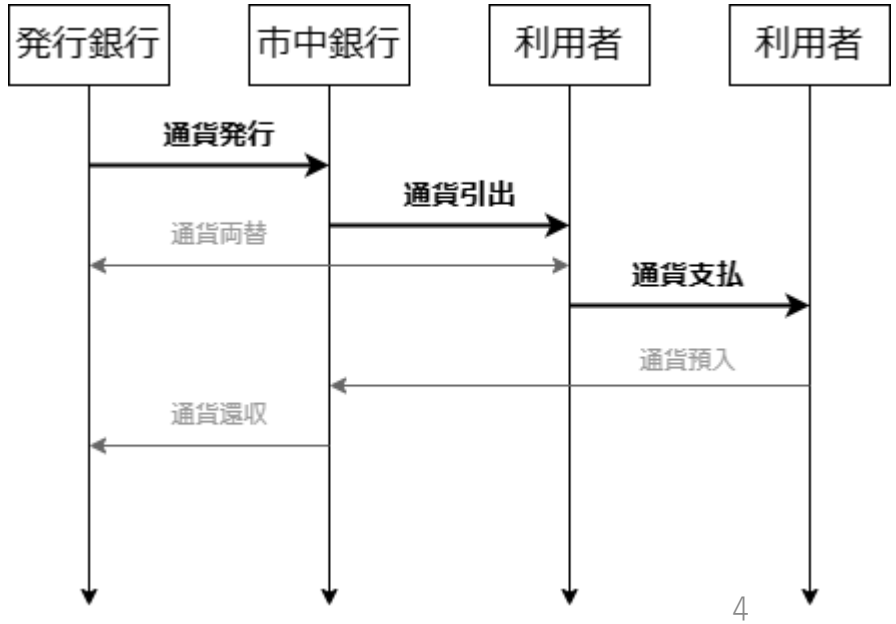
- 追跡不可能性 (Untraceability)
- 結合不可能性 (Unlinkability)
- 仮名性　(Pseudonymity)

通貨プロトコル概要

⓪事前準備(鍵生成, 証明書発行)

①通貨発行プロトコル
②通貨引出プロトコル
③通貨支払プロトコル
④通貨預入プロトコル
⑤通貨還収プロトコル
⑥通貨両替プロトコル

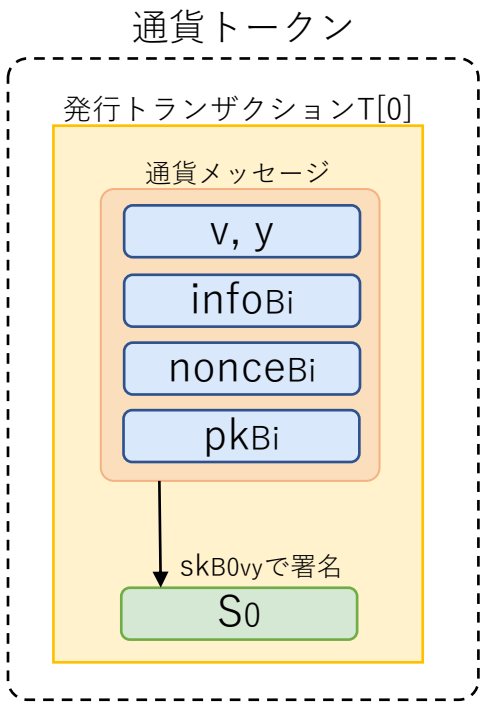




通貨発行プロトコル

通貨発行プロトコル

市中銀行からの発行要求に応じて通貨トークンを生成, 送付する



通貨メッセージ

(v, y, infoBi, pkBi, noceBi)

署名生成

S0 = Sign(v, y, infoBi, pkBi, noceBi)$_{skB0vy}$

通貨メッセージ

発行トランザクション

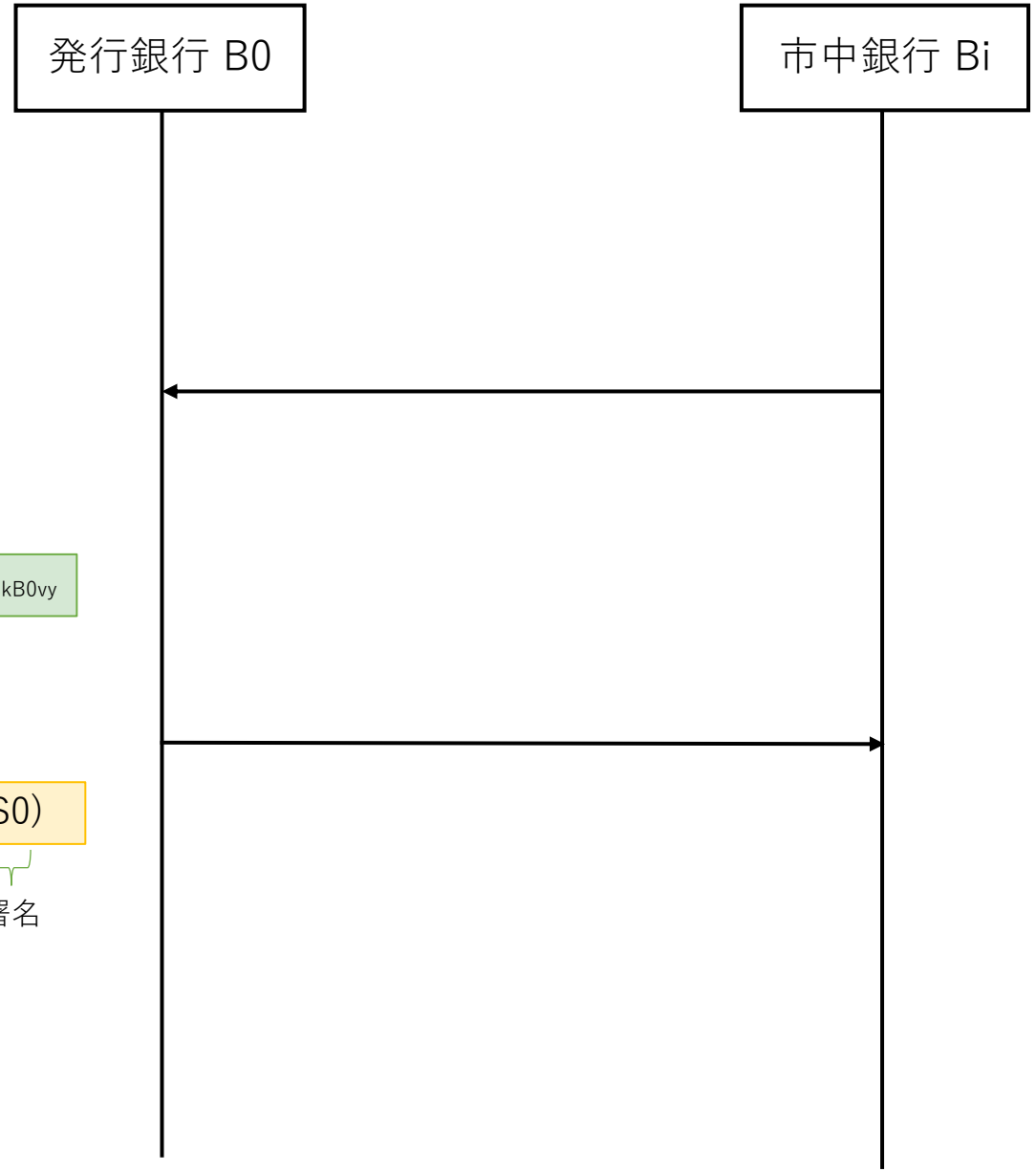
T[0] = ((v, y, infoBi, pkBi, noceBi), S0)

通貨メッセージ　署名

通貨トークン

(T[0])

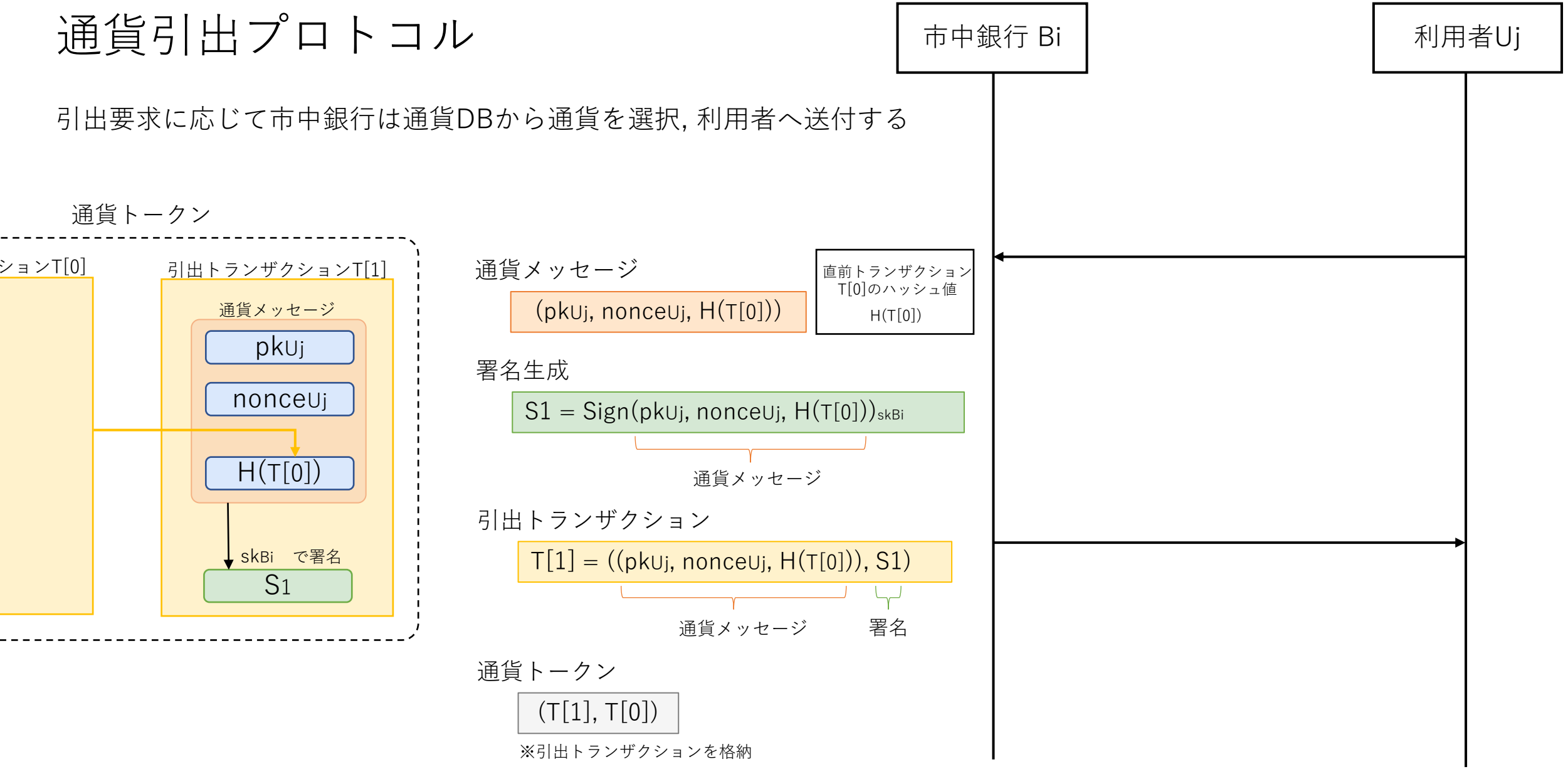
※発行トランザクションを格納



通貨引出プロトコル

通貨引出プロトコル

引出要求に応じて市中銀行は通貨DBから通貨を選択, 利用者へ送付する

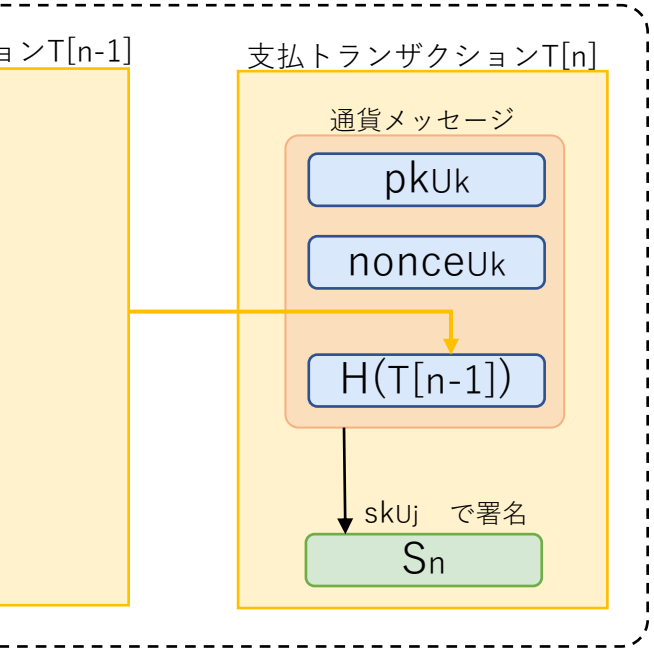


通貨支払プロトコル

通貨支払プロトコル

着金者が要求額を通知, 事前に支払額と釣銭額を確定して支払を実行する

通貨トークン



通貨メッセージ

(pkUk, nonceUk, H(T[n-1]))

署名生成

$Sn = Sign(pk_{Uk}, nonce_{Uk}, H(T[n-1]))_{skUj}$

通貨メッセージ

支払トランザクション

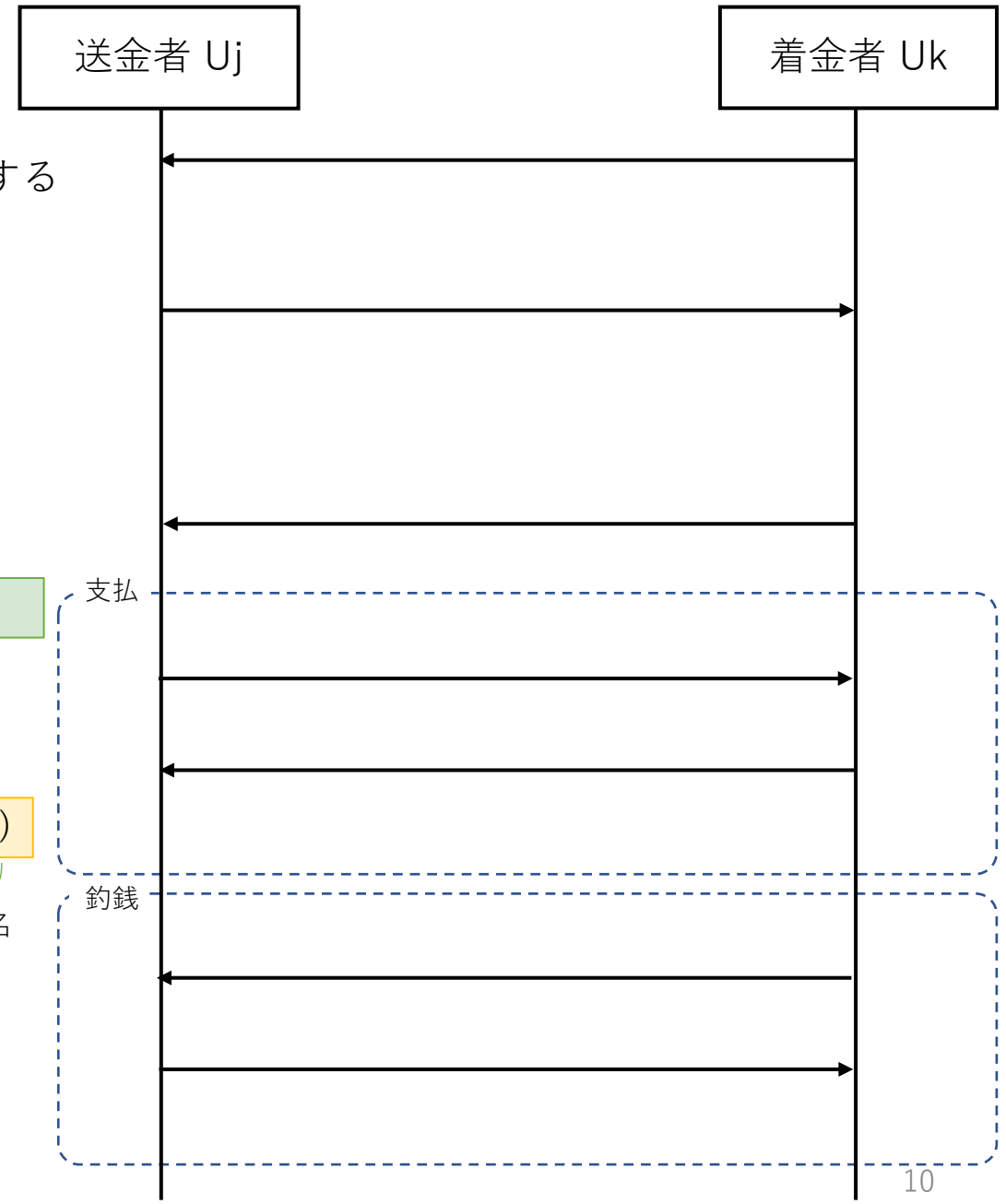
T[n] = ((pkUk, nonceUk, H(T[n-1])), Sn)

通貨メッセージ　署名

通貨トークン

(T[n], (T[n-1]~T[0]))

※引出トランザクションを格納

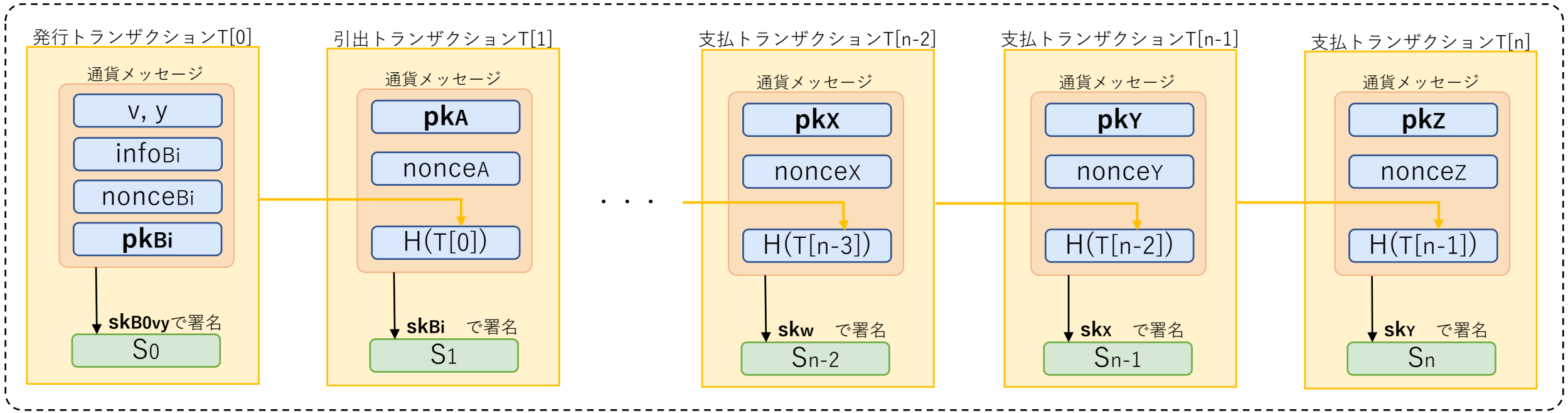


通貨署名の検証における工夫

通貨トークンに含まれるトランザクションの全てを検証する必要がある
- 含まれるトランザクションの数は変動する
- 通常の記述法では事前に定めた回数の署名検証しか実行できない

➡ 任意の長さの通貨トークンを検証するため, 専用の通貨検証プロセスを定義した

※「通貨メッセージに含まれる公開鍵」＝「次トランザクションの署名鍵の対となる公開鍵」
⇒トランザクションを辿って連鎖的に署名を検証することが可能



発行 : B0→Bi　引出 : Bi→A　支払 : W→X　支払 : X→Y　支払 : Y→Z

通貨署名の検証における工夫

通貨トークンは入子構造のタプル
- 左辺：新たに生成したトランザクション
- 右辺：直前までのトランザクション

(Tn, (Tn-1~T0))

最新のトランザクションから遡る形式で
トークンを分割しながら検証する

pkBi:市中銀行の公開鍵, S0:発行鍵による署名

T0＝(v, y, infBi, nonceBi, pkBi, S0) (発行)

pkUj:利用者Ujの公開鍵, S1:市中銀行Biによる署名

T1＝(pkUj, nonceUj, H(T[0]), S1) (引出)

pkUj:着金者Ukの公開鍵, S2:送金者Ujによる署名

T2＝(pkUk, nonceUk, H(T[n1]), S2) (支払)

⋮

pkU:着金者Xの公開鍵, Sn:送金者による署名

Tn＝(pkX, nonceX, H(T[n-1]), Sn) (支払)

通貨トークン

(T0)

(T1, T0**)**

(T2, (T1, T0)**)**

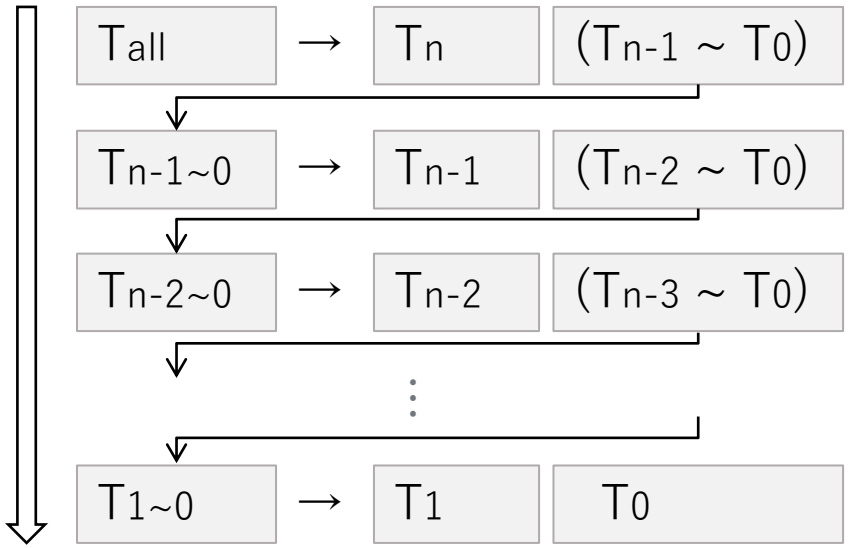
⋮

(Tn, (Tn-1, (Tn-2, …))**)**

検証処理の手順

1. トークンを分割
2. さらに分割できるか確認 (yes→3, no→4)
3. [2]で分割可能な場合
   →Tnを検証 , $T_{n-1\sim0}$を[1]へ
4. [2]で分割不可能な場合
   →T0, T1を検証 , 検証終了

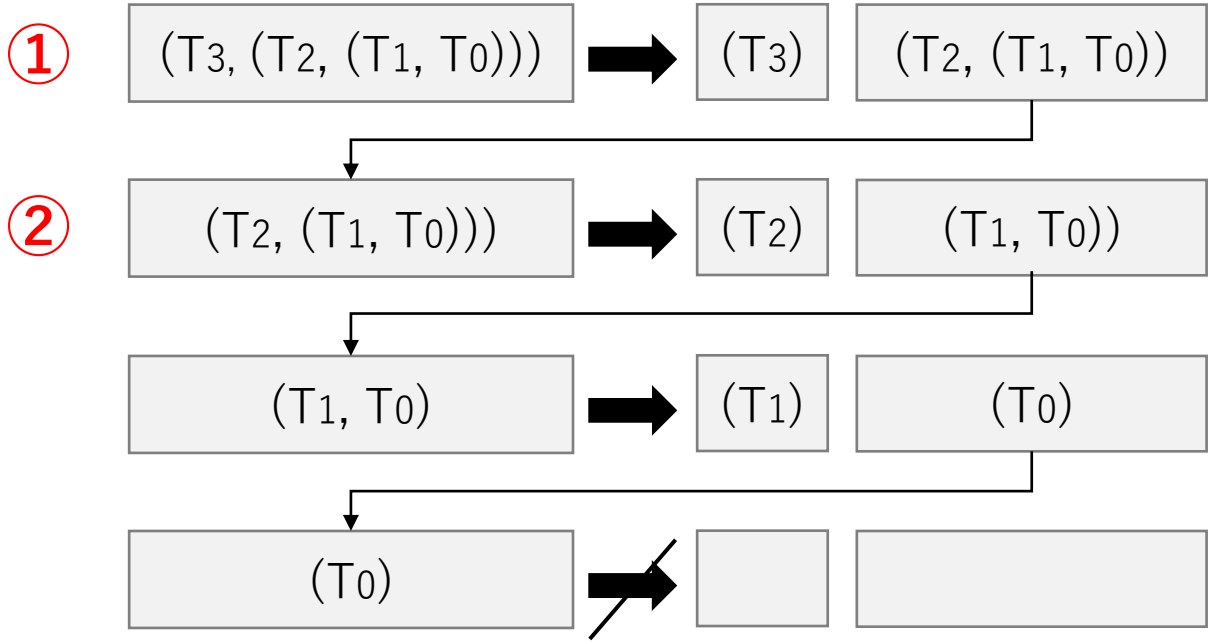
通貨トークン分割の様子



通貨署名の検証における工夫

検証処理の手順（例）

1. トークンを分割
2. さらに分割できるか確認 (yes→3, no→4)
3. 2で分割可能
   →Tnを検証 , $T_{n-1\sim0}$を1へ
4. 2で分割不可
   →T0, T1を検証 , 検証終了



③

$T_3 = (pk_C, nonce_C, H(T_2), \mathbf{S_3})$ Verif(S_3)$_{pk_B}$

$T_2 = (\mathbf{pk_B}, nonce_B, H(T_1), S_2)$ Verif(S_2)$_{pk_A}$

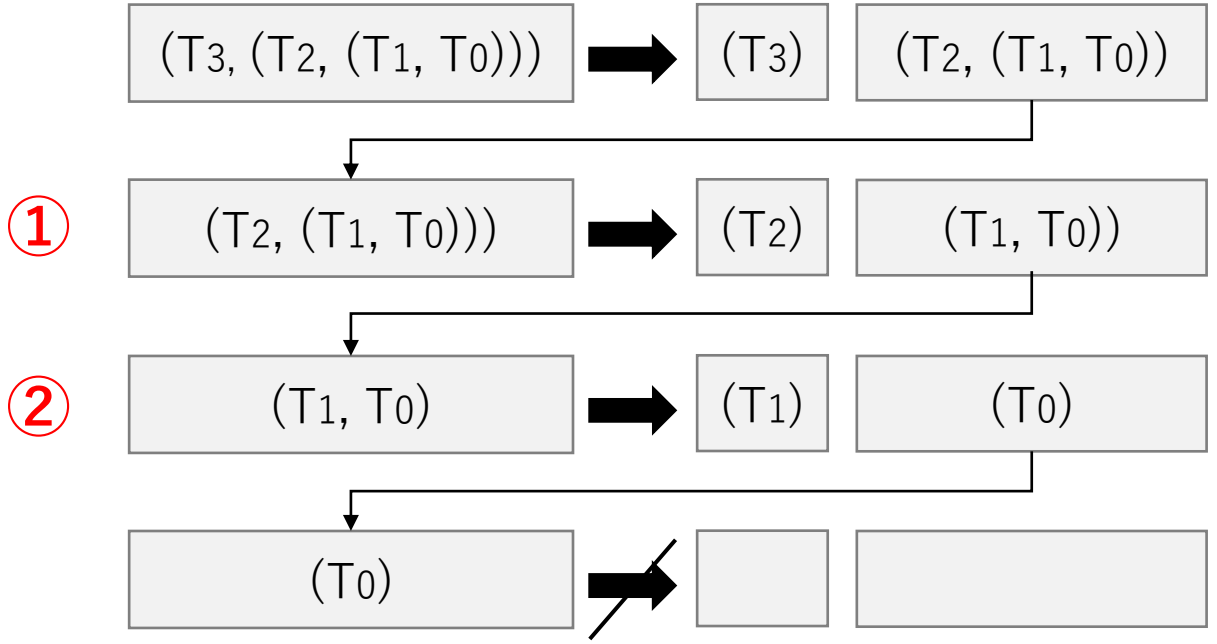
$T_1 = (pk_A, nonce_A, H(T_0), S_1)$ Verif(S_1)$_{pk_{Bi}}$

$T_0 = (v,y,info_{Bi},nonce_{Bi},pk_{Bi},S_0)$ Verif(S_0)$_{pk_{B0vy}}$

通貨署名の検証における工夫

検証処理の手順（例）

1. トークンを分割
2. さらに分割できるか確認 (yes→3, no→4)
3. 2で分割可能
   →Tnを検証 , $T_{n-1\sim0}$を1へ
4. 2で分割不可
   →T0, T1を検証 , 検証終了



③

$T_3 = (pk_C, nonce_C, H(T_2), S_3)$ $\quad Verif(S_3)_{pk_B}$

$T_2 = (pk_B, nonce_B, H(T_1), \mathbf{S_2})$ $\quad Verif(S_2)_{pk_A}$

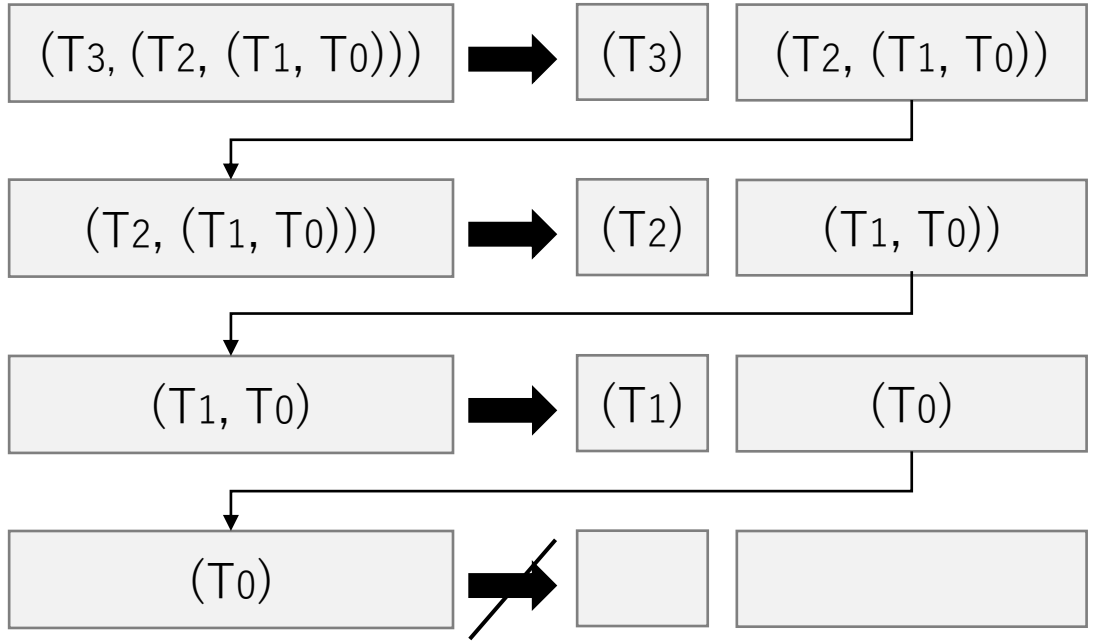
$T_1 = (\mathbf{pk_A}, nonce_A, H(T_0), S_1)$ $\quad Verif(S_1)_{pk_{Bi}}$

$T_0 = (v,y,info_{Bi},nonce_{Bi},pk_{Bi},S_0)$ $\quad Verif(S_0)_{pk_{B0vy}}$

通貨署名の検証における工夫

検証処理の手順（例）

1. トークンを分割
2. さらに分割できるか確認 (yes→3, no→4)
3. 2で分割可能
   →Tnを検証 , $T_{n-1\sim0}$を1へ
4. 2で分割不可
   →T0, T1を検証 , 検証終了



④

$T_3 = (pk_C, nonce_C, H(T_2), S_3)$ $Verif(S_3)_{pk_B}$

$T_2 = (pk_B, nonce_B, H(T_1), S_2)$ $Verif(S_2)_{pk_A}$

$T_1 = (pk_A, nonce_A, H(T_0), \mathbf{S_1})$ $Verif(S_1)_{pk_{Bi}}$

$T_0 = (v,y,info_{Bi},nonce_{Bi},\mathbf{pk_{Bi}},S_0)$ $Verif(S_0)_{pk_{B0vy}}$

※自己証明書の検証

通貨署名の検証（繰り返し処理）

通貨検証用プロセス

```
let CHECK_SIG =
  in(oc4,(Tpall'':bitstring));
  (
    let (Tn:bitstring,Tp_nm1:bitstring) = Tpall'' in
    let (Tnm1:bitstring,Tp_nm2:bitstring) = Tp_nm1 in
    (
       let (pkUy:spkey,Uy_nonce:bitstring,H_Tnm2:bitstring,Tp_nm2:bitstring) = Tnm1 in
       let (pkUx:spkey,Ux_nonce:bitstring,=H(Tnm1),Tp_nm1:bitstring) = Tn in
       if verif(Tp_nm1, (pkUx,Ux_nonce, H(Tnm1)), pkUy) = true then (* Tnの検証 *)

       out(oc4, (Tp_nm1))
       (* 分割可能なTp_nm1を再度流通、CHECK_SIGが再帰的に実行される *)
    )
    else
    (
       (* これ以上に分割できないTp_nm1はT0 *)
       let (v':bitstring,y':bitstring,Bi_info':bitstring,pkBi':spkey,Bi_nonce':bitstring,S0':bitstring) = Tp_nm1 in
       get vy_cert_table(=(v',y'), Cert_pkB0_v_y) in
       let (pkB0_v_y:spkey, sign_pkB0_v_y:bitstring) = Cert_pkB0_v_y in
       if verif(sign_pkB0_v_y,  spkey_to_bitstring(pkB0_v_y), pkB0_v_y) = true then (* 自己証明書の検証 *)
       if verif(S0', (v',y',Bi_info',pkBi',Bi_nonce'), pkB0_v_y) = true then (* S0の検証 *)

       (* これ以上に分割できないTp_nm1に対応するTnはT1 *)
       let (pkUj':spkey,Uj_nonce':bitstring,=H(Tp_nm1),S1':bitstring) = Tn in
       if verif(S1', (pkUj',Uj_nonce', H(Tp_nm1)), pkBi') = true then (* S1の検証 *)

       out(oc4, (signature_true))
    )
    )
  ).
```

着金者プロセス – 通貨検証部分

```
(* ===特殊処理(CHECK_SIG) start=== *)
out(oc4,(Tpall')); (* 支払通貨の検証開始 *)

in(oc4,(signature_flag:bitstring));
if (signature_flag = signature_true) then (* signature_flag = signature_true の場合，支払通貨の検証成功 *)
(* ===特殊処理(CHECK_SIG) end=== *)
```

通貨署名の検証（繰り返し処理）

通貨検証用プロセス

```
in(oc4,(Tpall'':bitstring));
let (Tn:bitstring,Tp_nm1:bitstring) = Tpall'' in
```

```
let (Tnm1:bitstring,Tp_nm2:bitstring) = Tp_nm1 in
(
   let (pkUy:spkey,Uy_nonce:bitstring,H_Tnm2:bitstring,Tp_nm2:bitstring) = Tnm1 in
   let (pkUx:spkey,Ux_nonce:bitstring,=H(Tnm1),Tp_nm1:bitstring) = Tn in
   if verif(Tp_nm1, (pkUx,Ux_nonce, H(Tnm1)), pkUy) = true then (* Tnの検証 *)

   out(oc4, (Tp_nm1))
   (* 分割可能なTp_nm1を再度流通、CHECK_SIGが再帰的に実行される *)
)
```

```
else
(
   (* これ以上に分割できないTp_nm1はT0 *)
   let (v':bitstring,y':bitstring,Bi_info':bitstring,pkBi':spkey,Bi_nonce':bitstring,S0':bitstring) = Tp_nm1 in
   get vy_cert_table(=(v',y'), Cert_pkB0_v_y) in
   let (pkB0_v_y:spkey, sign_pkB0_v_y:bitstring) = Cert_pkB0_v_y in
   if verif(sign_pkB0_v_y,  spkey_to_bitstring(pkB0_v_y), pkB0_v_y) = true then (* 自己証明書の検証 *)
   if verif(S0', (v',y',Bi_info',pkBi',Bi_nonce'), pkB0_v_y) = true then (* S0の検証 *)

   (* これ以上に分割できないTp_nm1に対応するTnはT1 *)
   let (pkUj':spkey,Uj_nonce':bitstring,=H(Tp_nm1),S1':bitstring) = Tn in
   if verif(S1', (pkUj',Uj_nonce', H(Tp_nm1)), pkBi') = true then (* S1の検証 *)

   out(oc4, (signature_true))
)
```

着金者プロセス – 通貨検証部分

```
(* ===特殊処理(CHECK_SIG) start=== *)
out(oc4,(Tpall')); (* 支払通貨の検証開始 *)
```

分割可能な場合
$T_{n-1\sim0}$を再度検証プロセスへ

分割不可能な場合
T1, T0を検証したのち
完了フラグを送信

```
in(oc4,(signature_flag:bitstring));
if (signature_flag = signature_true) then (* signature_flag = signature_true の場合，支払通貨の検証成功 *)
(* ===特殊処理(CHECK_SIG) end=== *)
```

検証

検証 1 – 秘匿性

秘密鍵と利用者情報の秘匿性をattackerクエリで検証

(*認証鍵の秘匿性*)
query attacker(skA0).
query attacker(skA1).

(*通貨発行鍵の秘匿性*)
query attacker(skB0_10000_2022).
query attacker(skB0_1000_2022).
query attacker(skB0_100_2022).
query attacker(skB0_10_2022).
query attacker(skB0_1_2022).
query attacker(skB0_0p1_2022).

(*通貨署名鍵の秘匿性*).
query attacker(skBi).
query attacker(skUj).
query attacker(skUk).

(*利用者情報の秘匿性*)
query attacker(Uj_info).
query attacker(Uk_info).

検証結果

全ての項目において”ture.”
すなわち攻撃なしが出力された

検証２ – 認証性

プロトコルごとの認証性をイベントクエリで検証

(* 発行プロトコル：B0↔Bi *)
query event(B0_end) ==> event(Bi_begin).
query event(Bi_end) ==> event(B0_begin).

false.
true.

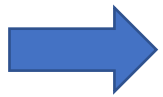
(* 引出プロトコル：Bi↔Uj *)
query event(Bi_end) ==> event(Uj_begin).
query event(Uj_end) ==> event(Bi_begin).

true.
true.

プライバシーの観点から，
ユーザ間認証は必ずしも必要でない．

（プロトコル設定としては問題なし）

(* 支払プロトコル：Uj↔Uk *)
query event(Uj_end) ==> event(Uk_begin).
query event(Uk_end) ==> event(Uj_begin).



can not be proved.
true.

検証結果

発行プロトコルB0 → Bi認証において“false.”が出力された
支払プロトコルUj → Uk認証において”can not be proved.”が出力された
その他の項目ではtrue. すなわち攻撃なしが出力された

検証２ – 認証性

ルート認証局の認証鍵の証明書　Cert(pkA0)
中間認証局の認証鍵の証明書　Cert(pkA1)
市中銀行の通貨署名鍵の証明書　Cert(pkBi)

出力結果の解釈

B0 → Biの認証 (発行プロトコル)

第三者がskA0で署名された公開鍵証明書を用いて発行依頼を送信した場合,
発行銀行の処理が正常に完了してしまうことが分かった.

・送金された通貨は対応する秘密鍵で署名することで支払に使用できる
・秘密鍵の秘匿性は確認済みであり, 攻撃者は署名用の秘密鍵を入手できない
→攻撃者は通貨を使用することができない. つまり攻撃としての意味を持たない.

※攻撃者によって何らかのロンダリングに類する振る舞いに使用される可能性は否定できない.

この攻撃については対応策が存在する.
以下のように条件式の記述を追記することで false→trueとなった　(攻撃が回避できることを確認)

追記

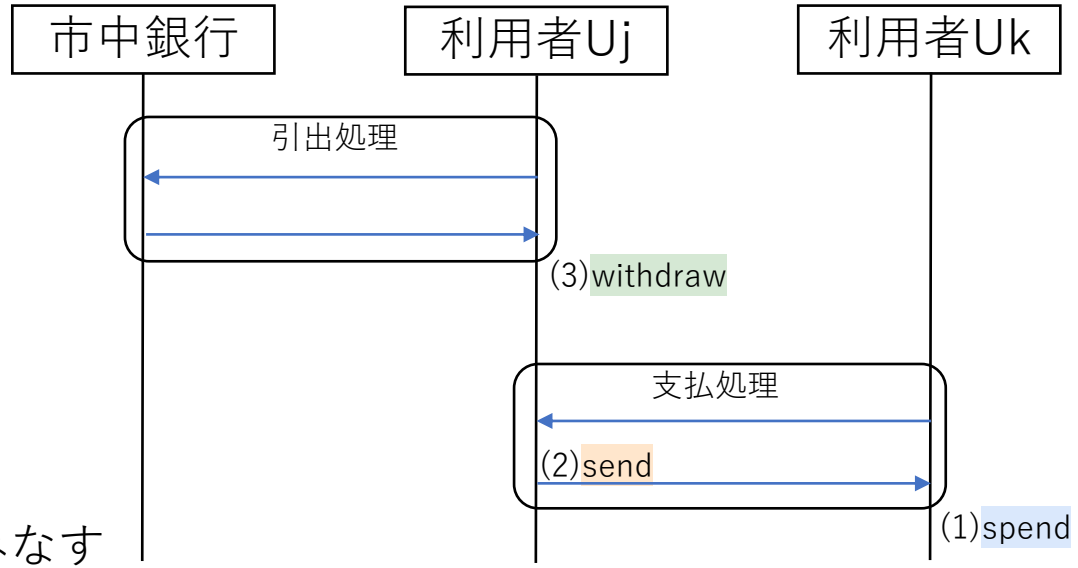
```
(* 発行依頼の受信 *)
in(B0Bi_ch, (Auth_pkBi:bitstring, (cnt:nat, v:bitstring, y:bitstring), Bi_nonce:bitstring));

if Auth_pkBi <> Auth_pkA1 then
if Auth_pkBi <> Auth_pkA0 then

(* 市中銀行の公開鍵証明書を検証 *)
let (pkBi:spkey, sign_pkBi:bitstring) = Auth_pkBi in
if verif(sign_pkBi, spkey_to_bitstring(pkBi), pkA0) = true then
```

検証３ – 通貨の偽造不可能性

偽装不可能性をイベントクエリで検証



クエリの設計…
・[支払いが完了](1) ==> [支払を開始している](2)
・[支払を開始](2) ==> [通貨を引出している](3)
上記の条件を満たす場合, 通貨は偽造されていないとみなす

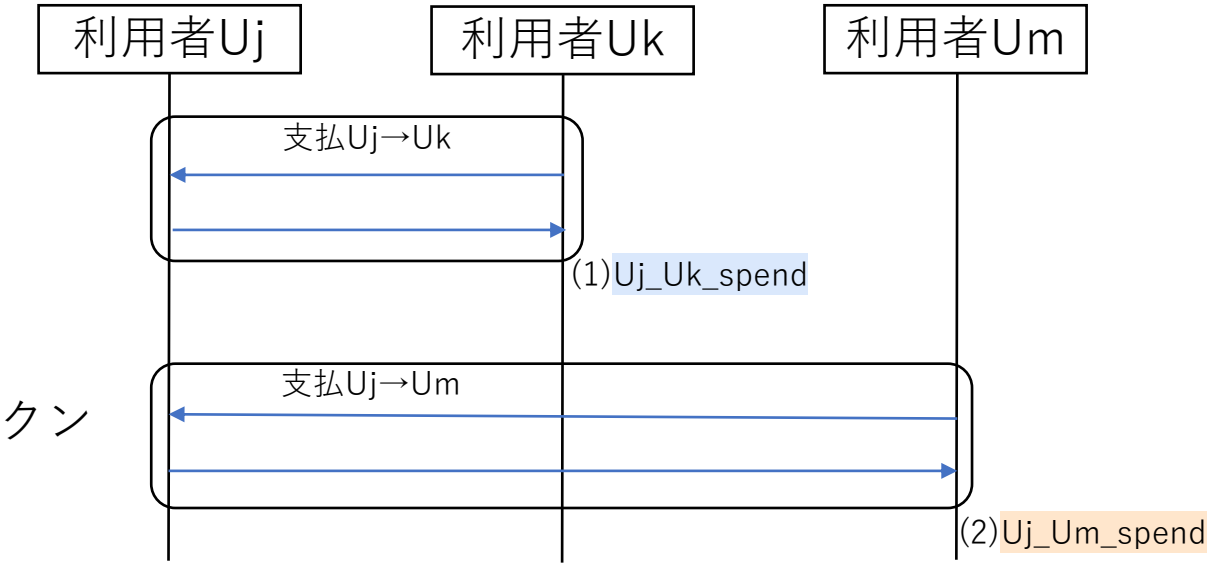
イベントの連続性を表現するため, イベントクエリを入子にして記述する
・[支払が完了](1) ==> [[支払を開始](2) ==> [通貨の引出し](3)]

```
(* 偽造不可能性 *)
query T0:bitstring,T1:bitstring,T2tn:bitstring;
 inj-event(end_Uj_Uk_spend(T2tn,T1,T0)) ==>
  (inj-event(begin_Uj_Uk_send(T2tn,T1,T0)) ==> inj-event(Bi_Uj_withdraw(T1,T0))).
```

検証結果

イベントクエリで検証した結果, “true.”
すなわち攻撃なしが出力された

検証４ – 二重使用



二重使用をイベントクエリで検証

クエリの設計…
　二つの支払い処理における支払完了時の通貨トークン

- Uj→Uk ：(Tn, (Tn-1~T0))(1)
- Uj→Um：(Tn', (Tn-1~T0)')(2)

「(Tn-1~T0) ≠ (Tn-1~T0)'」が成立するならば二重使用されていないとみなす

```
(* 二重使用 *)
query Tn:bitstring, Tn':bitstring, Tnm1:bitstring, Tnm1':bitstring;
 event(Uj_Uk_spend(Tn, Tnm1)) && event(Uj_Um_spend(Tn', Tnm1'))
  ==> Tnm1 <> Tnm1'.
```

検証結果

“can not be proved.”が出力され, attackトレースが出力された
オフライン型電子現金プロトコルでは二重使用を防止できないという理解と整合する
→通貨還収のタイミングで検知可能であるかを今後の実装で検証する予定.

まとめ

まとめ

解説した点

- 発行, 引出, 支払プロトコルの形式化について
- 偽造不可能性, 二重使用の検証クエリについて

工夫点

- 発行, 引出, 支払のプロトコルを一連の流れで1つのファイルに記述して検証を行った.
  →プロトコルどうしの組合せによる不具合がないかを検証
  →複数プロトコルを跨いでイベントを記述することで複雑な検証クエリの記述に成功
- private通信路を用いたプロセス間のパラメータ渡しの記述
  →別プロセス中のパラメータを他プロセスで利用することが可能に
- 特殊な繰り返し構造による通貨検証用プロセスを定義
  →ProVerifで任意の長さの通貨トークンを検証する機能の実現

今後の展望

第100回CSEC研究会 2023.3.7

“トークン型電子現金方式の二重使用検知およびプライバシーに関する形式検証の考察”[2]

・両替プロトコル, 預入プロトコル, 還収プロトコルの形式化
・二重使用検知の検証
・観測等価性の評価による仮名性の検証

今後更に…

・二重使用に関連した免責性についての検証
・その他プライバシー要件についての検証
・新たに着想を得たマネーロンダリングについての検証クエリを検討

[2] 奥田哲矢, 荒井研一, 齋藤恆和, 千田浩司, 中林美郷, 山村和輝, 宮澤俊之, 阿部正幸, “トークン型電子現金方式の二重使用検知およびプライバシー に関する形式検証の考察”, 情報処理学会研究報告書, Vol.2022-CSEC-100

ご清聴ありがとうございました