Intel SGXを利用した 効率的に入力制御可能な複数入力関数型暗号

田宮 寛人[1]　Enkhtaivan Batnyam[1]　Dhomse Pooja[1]　一色 寿幸[1]

概要：関数型暗号は暗号化されたデータに関する情報を秘匿したまま，データの関数値を計算できる暗号方式である．Fisch らは Intel SGX を利用することにより実用的な関数型暗号を構成できることを報告した．Fisch らの方式の拡張として，吉野らは関数の計算に用いられるデータが，信頼できる第三者（TTP）によって許可されたユーザ（データ提供者）のデータであるときに関数値を計算できる関数型暗号を提案した．吉野らの方式では，TTP と各ユーザが共有する乱数に基づいて入力制御が行われるため，乱数の管理コストが発生する．本稿では，Intel SGX を利用した TTP と各ユーザが乱数を共有せずに入力制御可能な複数入力関数型暗号を提案する．

キーワード：関数型暗号，Intel SGX，入力制御

Multi-Input Functional Encryption Using Intel SGX with Efficient Input Control

Hiroto Tamiya[1]　Enkhtaivan Batnyam[1]　Dhomse Pooja[1]　Toshiyuki Isshiki[1]

Abstract: Functional encryption is an encryption scheme which can calculate a function value of encrypted input data without revealing information about the data itself. Fisch et al. proposed a practical functional encryption scheme by utilizing Intel SGX. As an extension of their work, Yoshino et al. presented an encryption scheme which calculates the function value only when the input data is authorized by a trusted third party (TTP). Since this scheme confirms the authorization by using random numbers shared between the TTP and each user, the TTP and each user need to manage random numbers securely. In this work, we propose a functional encryption scheme using Intel SGX without sharing random numbers between the TTP and each user.

Keywords: Functional encryption, Intel SGX, input control

1. はじめに

関数型暗号は暗号化されたデータに関する情報を秘匿したまま，データの関数値を計算できる暗号方式である [1,3]．ハードウェアを用いないモデルでは，任意の関数に対する関数型暗号は識別不可能難読化などの非実用的な技術を用いてしか実現されていない [3]．一方で，Fisch，Vinayagamurthy，Boneh，Gorbunov は Intel SGX をモデル化したハードウェアモデルを用いることで任意の関数に対する実用的な関数型暗号を提案している [2]．

復号者が任意のデータを入力とする関数値を得ることができるとき，複数の関数値からデータに関する情報が漏洩する可能性がある．例えばデータ a，b，c の平均値とデータ a，b の平均値からデータ c の値が漏洩する．このような漏洩を防ぐために，吉野，手塚，品川，田中は信頼できる第三者（TTP）が関数への入力を制御可能な関数型暗号を提案している [5]．

吉野らの方式では TTP と各ユーザが共有する乱数に基

[1] 日本電気株式会社，〒 211-8666 神奈川県川崎市中原区下沼部 1753, NEC Corporation, 1753 Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8666, Japan. {h-tamiya@ay, b-enkhtaivan@bc, p-dhomse@cj, t-issiki@bx}.jp.nec.com

づいて入力制御が行われる．したがって，TTP とユーザに乱数を秘密に管理するコストが生じる．

そこで本稿では，TTP と各ユーザ（データ提供者）で乱数の共有が不要な Intel SGX を用いた複数入力関数型暗号を提案する．提案方式では，ユーザが生成した暗号文を利用して入力制御を行う．

2. 定義

本章では，Intel SGX をモデル化したハードウェアモデルおよび提案方式で利用する暗号プリミティブとその安全性を定義する．

2.1 表記

有限集合 X から要素 x を一様ランダムに選ぶことを $x \xleftarrow{r} X$ で表し，集合 $\{1, 2, \ldots, n\}$ を $[n]$ で表す．また，x と y を結合した列を $x||y$ で表す．さらに，x を入力とする確率的アルゴリズム $\mathcal{A}$ が y を出力することを $y \leftarrow \mathcal{A}(x)$ で表し，確率的多項式時間アルゴリズムを PPT で表す．そして，λ に関して無視できる値を $negl(\lambda)$ で表す．

2.2 Intel SGX [4]

Intel SGX の主要な機能である Isolation，Sealing，Attestation について概説する．

Isolation 機密なコードを Enclave と呼ばれるセキュア領域内で実行する機能である．悪意のある OS を含む外部からは Enclave 内のコードやデータにアクセスできない．

Sealing Enclave 内の秘密情報を Enclave 固有の鍵で暗号化し，Enclave 外に記憶する機能である．

Attestation Enclave 内で動作しているプログラムやプログラム実行時の入出力を検証する機能である．同じ環境上の Enclave により検証する Local Attestation と，異なる環境において誰でも検証することができる Remote Attestation がある．

2.3 ハードウェアモデル

Fisch らは Intel SGX をモデル化したハードウェア (HW) モデルを定義した [2]．本稿でも Fisch らが定義した HW モデルを用いる．

定義 1 (HW モデル [2])**.** HW モデルは (HW.Setup, HW.Load, HW.Run, HW.Run&Report, HW.Run&Quote, HW.ReportVerify, HW.QuoteVerify) の 7 つのインタフェースから成る．さらに，HW は内部状態として 2 つの変数 HW.sk_{report} と HW.sk_{quote} およびテーブル T を持つ．HW.sk_{report} と HW.sk_{quote} は鍵の保管に利用され，T は HW にロードされた Enclave プログラムの内部状態を扱うために利用される．

- HW.Setup(1^λ): HW を初期化する．HW.Setup はセキュリティパラメータ λ を入力として受け取り，秘密鍵 sk_{report} と sk_{quote} を生成し，それぞれ HW.sk_{report} と HW.sk_{quote} に記憶する．最後に公開パラメータ $params$ を生成し出力する．
- HW.Load($params, Q$): ステートフルなプログラムを Enclave にロードする．HW.Load は公開パラメータ $params$ とプログラム Q を入力として受け取り，Enclave を作成しプログラム Q をロードし，プログラム Q が動作している Enclave を特定するためのハンドル hdl を作成する．最後に，エントリ $T[hdl] = \emptyset$ として初期化し，hdl を出力する．
- HW.Run(hdl, in): Enclave にロードされているプログラムを実行する．HW.Run はステートフルなプログラム Q がロードされている Enclave に対応するハンドル hdl とプログラム Q への入力 in を受け取り，状態 $T[hdl]$ のプログラム Q を実行し，出力 out を記憶する．最後に，状態 $T[hdl]$ を更新し，出力 out を出力する．
- HW.Run&Report(hdl, in): Enclave にロードされているプログラムを実行し，さらに同じ HW プラットフォームで検証が実行される出力に対する検証情報 (Report) を生成する．HW.Run&Report はステートフルなプログラム Q がロードされている Enclave に対応するハンドル hdl とプログラム Q への入力 in を受け取り，状態 $T[hdl]$ のプログラム Q を実行し，出力 out を記憶し，状態 $T[hdl]$ を更新する．その後，$report := (md_{hdl}, tag_Q, in, out, mac)$ を出力する．ここで，md_{hdl} は hdl に対応する Enclave のメタデータ，tag_Q は Enclave の中で動作しているプログラム Q を特定するためのタグ (例えば，プログラム Q のハッシュ値)，mac は sk_{report} を利用した $(md_{hdl}, tag_Q, in, out)$ に対するメッセージ認証子（MAC) である．
- HW.Run&Quote(hdl, in): Enclave にロードされているプログラムを実行し，さらにパブリックに検証可能な出力に対する検証情報 (Quote) を生成する．HW.Run&Quote はステートフルなプログラム Q がロードされている Enclave に対応するハンドル hdl とプログラム Q への入力 in を受け取り，状態 $T[hdl]$ のプログラム Q を実行し，出力 out を記憶し，状態 $T[hdl]$ を更新する．その後，$quote := (md_{hdl}, tag_Q, in, out, \sigma)$ を出力する．ここで，md_{hdl} は hdl に対応する Enclave のメタデータ，tag_Q は Enclave の中で動作しているプログラム Q を特定するためのタグ，σ は sk_{quote} を利用した $(md_{hdl}, tag_Q, in, out)$ に対する署名である．
- HW.ReportVerify($hdl', report$): Report を検証する．検証を行う Enclave に対応するハンドル hdl' と $report = (md_{hdl}, tag_Q, in, out, mac)$ を受け取り，sk_{report} を利用して mac を検証する．もし有効であれ

ば 1 を出力し，$T[hdl']$ に $(report, 1)$ を追加する．そうでなければ，0 を出力し，$T[hdl']$ に $(report, 0)$ を追加する．

- HW.QuoteVerify$(params, quote)$: Quote を検証する．$params$ と $quote = (md_{hdl}, tag_Q, in, out, \sigma)$ を入力として受け取り，σ に対する署名検証に成功すれば 1 を，そうでなければ 0 を出力する．

また，プログラム空間 $\mathcal{Q}$ とハンドル空間 $\mathcal{H}$ において，任意の $Q \in \mathcal{Q}$，Q の入力域に含まれる任意の in，任意のハンドル $hdl, hdl' \in \mathcal{H}$ に対して，

- Run の正当性: $out = Q(in)$ である．
- Report と ReportVerify の正当性:
  $report =$ HW.Run&Report(hdl, in) として，

$$\Pr[\text{HW.ReportVerify}(hdl', report) = 0] \leq negl(\lambda)$$

- Quote と QuoteVerify の正当性:
  $quote =$ HW.Run&Quote(hdl, in) として，

$$\Pr[\text{HW.QuoteVerify}(params, quote) = 0] \leq negl(\lambda)$$

を満たす．

定義 2 (Report の偽造不可能性 [2])**.** Report の偽造不可能性は MAC の偽造不可能性と同様に定義される．挑戦者 $\mathcal{C}$ と攻撃者 $\mathcal{A}$ による次のゲームを考える．

(1) $\mathcal{C}$ は HW.Setup(1^λ) を動作させ，得られた公開パラメータ $params$ を $\mathcal{A}$ に送る．
(2) $\mathcal{C}$ はリスト $query$ を $query = \emptyset$ と初期化する．
(3) $\mathcal{A}$ は任意の $(params, Q)$ に対して HW.Load$(params, Q)$ を動作させ，その出力 hdl を得る．
(4) $\mathcal{A}$ は任意の (hdl, in) に対して HW.Run&Report(hdl, in) を動作させ，その出力 $report = (md_{hdl}, tag_Q, in, out, mac)$ を得る．動作ごとに C はリスト $query$ に $(md_{hdl}, tag_Q, in, out)$ を追加する．
(5) $\mathcal{A}$ は任意の $(hdl', report)$ に対して，HW.ReportVerify$(hdl', report)$ を動作させ，その出力を得る．
(6) 最後に $\mathcal{A}$ は $report^* = (md^*_{hdl}, tag^*_Q, in^*, out^*, mac^*)$ を出力する．

あるハンドル hdl'^* に対して次の条件を満たすとき，攻撃者はゲームに勝利すると呼ぶ．

(1) HW.ReportVerify$(hdl'^*, report^*) = 1$
(2) $(md^*_{hdl}, tag^*_Q, in^*, out^*) \notin query$

上記のゲームにセキュリティパラメータ λ に関して無視できない確率で勝利する PPT 攻撃者 $\mathcal{A}$ が存在しないとき，HW は Report が偽造不可能という．

定義 3 (Quote の偽造不可能性 [2])**.** Quote の偽造不可能性は署名の偽造不可能性と同様に定義される．挑戦者 $\mathcal{C}$ と攻撃者 $\mathcal{A}$ による次のゲームを考える．

(1) $\mathcal{C}$ は HW.Setup(1^λ) を動作させ，得られた公開パラメータ $params$ を $\mathcal{A}$ に送る．
(2) $\mathcal{C}$ はリスト $query$ を $query = \emptyset$ と初期化する．
(3) $\mathcal{A}$ は任意の $(params, Q)$ に対して HW.Load$(params, Q)$ を動作させ，その出力 hdl を得る．
(4) $\mathcal{A}$ は任意の (hdl, in) に対して HW.Run&Quote(hdl, in) を動作させ，その出力 $quote = (md_{hdl}, tag_Q, in, out, \sigma)$ を得る．動作ごとに C はリスト $query$ に $(md_{hdl}, tag_Q, in, out)$ を追加する．
(5) 最後に $\mathcal{A}$ は $quote^* = (md^*_{hdl}, tag^*_Q, in^*, out^*, \sigma^*)$ を出力する．

次の条件を満たすとき，攻撃者はゲームに勝利すると呼ぶ．

(1) HW.ReportVerify$(hdl'^*, quote^*) = 1$
(2) $(md^*_{hdl}, tag^*_Q, in^*, out^*) \notin query$

上記のゲームにセキュリティパラメータ λ に関して無視できない確率で勝利する PPT 攻撃者 $\mathcal{A}$ が存在しないとき，HW は Quote が偽造不可能という．

2.4 暗号プリミティブ

提案方式を構成する暗号プリミティブとして，公開鍵暗号，署名，ハッシュ関数を定義する．

定義 4 (公開鍵暗号)**.** 公開鍵暗号方式 PKE はアルゴリズムの組 (PKE.KeyGen, PKE.Enc, PKE.Dec) である．

- PKE.KeyGen(1^λ): セキュリティパラメータ λ を入力として受け取り，暗号化鍵と復号鍵の組 (ek, dk) を出力する．
- PKE.Enc(ek, m): 暗号化鍵 ek と平文 m を入力として受け取り，暗号文 ct を出力する．
- PKE.Dec(dk, ct): 復号鍵 dk と暗号文 ct を入力として受け取り，復号した平文 m または $\perp$ を出力する．

また，公開鍵暗号方式 PKE は任意のセキュリティパラメータ $\lambda \in \mathbb{N}$，任意の平文 $m \in \mathcal{M}$，$(ek, dk) \leftarrow$ PKE.KeyGen(1^λ) に対して，

$$\Pr[\text{PKE.Dec}(dk, \text{PKE.Enc}(ek, m)) \neq m] \leq negl(\lambda)$$

を満たす．

定義 5 (公開鍵暗号の IND-CCA2 安全性)**.** 公開鍵暗号方式 PKE の IND-CCA2 安全性は挑戦者 $\mathcal{C}$ と攻撃者 $\mathcal{A}$ による以下のゲームを用いて定義される．

(1) $\mathcal{C}$ は PKE.KeyGen(1^λ) を動作させ，(ek, dk) を得る．その後，ek を $\mathcal{A}$ に送る．
(2) $\mathcal{A}$ は ct を適応的に選び，PKE.Dec(dk, ct) の結果を知ることができる．
(3) $\mathcal{A}$ は m_0 と m_1 を $\mathcal{C}$ に送る．
(4) $\mathcal{C}$ は $b \xleftarrow{r} \{0, 1\}$ を選び，チャレンジ暗号文 $ct \leftarrow$

PKE.Enc(ek, m_b) を得る．その後，ct^* を A に送る．

(5) $\mathcal{A}$ は $ct \neq ct^*$ を適応的に選び，PKE.Dec(dk, ct) の結果を知ることができる．

(6) $\mathcal{A}$ は b' を出力する．

攻撃者 $\mathcal{A}$ の優位性を

$$\mathrm{Adv}_{pke}(\mathcal{A}) = \left| \Pr[b' = b] - \frac{1}{2} \right|$$

とする．λ に関して無視できない優位性を持つ PPT 攻撃者 A が存在しないとき，公開鍵暗号方式 PKE は IND-CCA2 安全であるという．

定義 6 (署名)．署名方式 S はアルゴリズムの組 (S.KeyGen, S.Sign, S.Verify) である．

- S.KeyGen(1^λ): セキュリティパラメータ λ を入力として受け取り，署名鍵と検証鍵の組 (sk, vk) を出力する．
- S.Sign(sk, m): 署名鍵 sk と平文 m を入力として受け取り，m に対する sk を用いた署名 σ を出力する．
- S.Verify(vk, σ, m): 検証鍵 vk，署名 σ，平文 m を入力として受け取り，署名が有効なとき 1 を，そうでないとき 0 を出力する．

また，署名方式 S は任意セキュリティパラメータ $\lambda \in \mathbb{N}$，任意の平文 $m \in \mathcal{M}$，$(sk, vk) \leftarrow$ S.KeyGen(1^λ) に対して，

$$\Pr[\text{S.Verify}(vk, \text{S.Sign}(sk, m), m) = 0] \leq negl(\lambda)$$

を満たす．

定義 7 (署名の EUF-CMA 安全性)．署名方式 S の EUF-CMA 安全性は挑戦者 $\mathcal{C}$ と攻撃者 $\mathcal{A}$ による以下のゲームを用いて定義される．

(1) $\mathcal{C}$ は S.KeyGen(1^λ) を動作させ，(sk, vk) を得る．その後，vk を $\mathcal{A}$ に送る．

(2) $\mathcal{C}$ はリスト $query$ を $query = \emptyset$ と初期化する．

(3) $\mathcal{A}$ は平文 m を適応的に選び，S.Sign(sk, m) の結果を知ることができる．このとき，C は m を $query$ に追加する．

(4) $\mathcal{A}$ は平文と署名の組 (m^*, σ^*) を出力する．ただし，$m^* \neq m$ でなければならない．

攻撃者 $\mathcal{A}$ の優位性を

$$\mathrm{Adv}_S(\mathcal{A}) = \Pr[\text{S.Verify}(vk, \sigma^*, m^*) = 1]$$

とする．λ に関して無視できない優位性持つ PPT 攻撃者 A が存在しないとき，署名方式 S は EUF-CMA 安全であるという．

定義 8 (衝突困難なハッシュ関数)．関数族 $\mathrm{H} = \{H_i\}$ と関数選択アルゴリズム H.Gen の組は，次を満たすとき衝突困難なハッシュ関数であるという．任意の $\lambda \in \mathbb{N}$，任意の PPT 攻撃者 $\mathcal{A}$ に対し，$H \leftarrow$ H.Gen(1^λ)，$(x, y) \leftarrow \mathcal{A}(H)$ として

$$\Pr[H(x) = H(y)] \leq negl(\lambda)$$

3. 入力制御可能な関数型暗号

本章では，入力制御可能な関数型暗号とその安全性を定義する．

3.1 定義

吉野らは各ユーザが異なる暗号化鍵を利用して平文を暗号化する関数型暗号を定義している [5]．一方で，我々はすべてのユーザが同じ暗号化鍵で平文を暗号化する関数型暗号を考えている．そのため，吉野らの定義とは異なる入力制御可能な関数型暗号を定義する．

以下で定義する入力制御可能な関数型暗号は Fisch らのが定義した関数型暗号 [2] に，入力制御を行うためのトークンの生成を行う Tokengen アルゴリズムを追加し，さらに複数入力関数型暗号に拡張したものである．

暗号化アルゴリズム以外は HW オラクルにアクセスできる．ただし，復号前の準備として復号者は鍵発行機関に HW に対する操作を行ってもらう必要があり，この操作を KM オラクルとして表記する．更に，この操作は復号する前にのみ実行すれば良いことを明示するため，復号アルゴリズムを DecSetup と Dec に分け，DecSetup のみ KM オラクルにアクセスできる制限している．

定義 9 (入力制御可能な関数型暗号)．入力制御可能な関数型暗号方式 ICFE はアルゴリズムの組 (ICFE.Setup, ICFE.Enc, ICFE.DecKeygen, ICFE.Tokengen, ICFE.DecSetup, ICFE.Dec) である．

- ICFE.Setup$^{\mathrm{HW}}(1^\lambda)$: セットアップアルゴリズムである．セキュリティパラメータ λ を入力として受け取り，マスター秘密鍵 msk とマスター公開鍵 mpk を出力する．
- ICFE.Enc(mpk, x_k): 暗号化アルゴリズムである．ユーザ k はマスター公開鍵 mpk と平文 x_k を入力として受け取り，暗号文 ct_k を出力する．
- ICFE.DecKeygen$^{\mathrm{HW}}(msk, P)$: 復号鍵生成アルゴリズムである．マスター秘密鍵 msk と計算したい関数 f を実装したプログラム P を入力として受け取り，復号鍵 sk_P を出力する．
- ICFE.Tokengen$^{\mathrm{HW}}(msk, P, S)$: トークン生成アルゴリズムである．マスター秘密鍵 msk とプログラム P，順序付きユーザリスト $S = \{i_1, i_2, \ldots, i_m\}$ を受け取り，$token_{P,S}$ を出力する．
- ICFE.DecSetup$^{\mathrm{KM,HW}}(mpk)$: 復号セットアップアルゴリズムである．マスター公開鍵 mpk を入力として，ICFE.Dec で使う為のハンドル hdl を出力する．
- ICFE.Dec$^{\mathrm{HW}}(hdl, sk_P, token_{P,S}, ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m})$: 復号アルゴリズムである．ハンドル hdl とプログラム P に関する復号鍵 sk_P，P と順序付きユーザリスト S に関するトークン $token_{P,S}$，暗号文 $ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m}$

を入力として受け取り，$S = \{i_1, i_2, \ldots, i_m\}$ のとき，$f(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ を出力する．そうでなければ，$\perp$ を出力する．ただし，x_{i_k} は ct_{i_k} の平文である．

また，プログラムの空間を $\mathcal{P}$，平文の空間を $\mathcal{M}$ において，任意の $\lambda \in \mathbb{N}$ と任意の $P \in \mathcal{P}$，任意の $m \in \mathbb{N}$，任意の $S = \{i_1, i_1, \ldots, i_m\}$，任意の $x_i \in \mathcal{M}$ に対して，

$$(msk, mpk) \leftarrow \text{ICFE.Setup}^{\text{HW}}(1^\lambda)$$
$$ct_k \leftarrow \text{FE.Enc}(mpk, x_k) \quad for\ k \in [m]$$
$$sk_P \leftarrow \text{FE.DecKeygen}^{\text{HW}}(msk, P)$$
$$token_{P,S} \leftarrow \text{FE.Tokengen}^{\text{HW}}(msk, P, S)$$
$$hdl \leftarrow \text{FE.DecSetup}^{\text{KM,HW}}(mpk)$$
$$y \leftarrow \text{FE.Dec}^{\text{HW}}(hdl, sk_P, token_{P,S}, ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m})$$

とする．このとき

$$\Pr[y \neq P(x_1, x_2, \ldots, x_m)] \leq negl(\lambda)$$

を満たす．

3.2 安全性

吉野ら [5] は Goldwasser ら [3] が定義した安全性を参考に入力制御可能な関数型暗号の安全性を定義した．本稿では吉野らが定義した安全性を参考に，定義 9 の入力制御可能な関数型暗号に対するシミュレーションベース安全性を定義する．

安全性はチャレンジ平文の個数 q によってパラメータ付けられる．

定義 10 (q-シミュレーション安全性)**.** 任意のステートフルな PPT 攻撃者 $\mathcal{A}$ に対し，あるステートフルなシミュレータ $\mathcal{S}$ が存在し，図 1 の Experiment $\text{REAL}_{\mathcal{A}}^{\text{ICFE}}(1^\lambda)$ と Experiment $\text{IDEAL}_{\mathcal{A},\mathcal{S}}^{\text{ICFE}}(1^\lambda)$ の出力が計算量的に識別不可能なとき，入力制御可能な関数型暗号 ICFE は q-シミュレーション安全であるという．

4. 提案方式

本章では提案する入力制御可能な関数型暗号方式について説明する．

吉野らの方式 [5] では TPP とユーザが共有した乱数に基づいて入力制御が行われる．したがって，鍵管理者とユーザにとって乱数を管理するコストが発生する．一方で，提案方式ではユーザが生成した暗号文に基づいて入力制御が行われる．よって，TPP とユーザが乱数を管理するコストは発生しない．

提案方式の概要を図 **2** に示す．ただし簡単のため，実際の提案方式から省力している記述や動作が存在する．

提案方式では，復号者は関数に入力したいデータに対応するユーザリストを TTP に送信し，TTP は受け取ったユーザリストに対応する暗号文をデータベースから取得する．その後，TTP は取得した暗号文のハッシュ値と計算したい関数を実装した Enclave プログラムのタグに対する署名をトークンとして生成し，復号者に送信する．復号者は受け取ったトークンと暗号文を Enclave プログラムに入力する．もし Enclave 内でトークンの検証に成功すれば，暗号文を復号し平文を得て，それらを入力とする関数を計算する．上述のように，提案方式では TTP が生成するタグと暗号文に対する署名の偽造困難性を利用して，TTP による入力制御を可能としている．

以下に提案方式の詳細なアルゴリズムを記述する．PKE を公開鍵暗号，S を署名，H を衝突困難なハッシュ関数とする．以下でハードウェアにロードするプログラム Q_{KME} と Q_{DE}，Q_{FE} を定義する．Q_{KME} はハードウェア KM に，Q_{DE} と Q_{FE} はハードウェア DN にロードされる．また，前処理としてハードウェア KM とハードウェア DN は HW.Setup(1^λ) を実行し，その出力 $params_{\text{KM}}$，$params_{\text{DN}}$ をそれぞれ出力する．

Q_{KME}:

- 入力 ("$init$", 1^λ) に対して，
  (1) $(ek, dk) \leftarrow \text{PKE.KeyGen}(1^\lambda)$，
  $(sk, vk) \leftarrow \text{S.KeyGen}(1^\lambda)$ を動作させる．
  (2) ステート $state$ を (ek, dk, sk, vk) に更新し，(ek, vk) を出力する．
- 入力 ("$provision$", $quote, params$) に対して，
  (1) $quote = (md_{hdl}, tag_Q, in, out, \sigma)$ とし，$tag_Q \neq tag_{DE}$ なら $\perp$ を出力する．ただし，tag_{DE} は Q_{DE} のタグであり，Q_{KME} にハードコーディングされている．
  (2) $in =$ ("$init\ setup$", vk) と分解し，vk が $state$ に記憶されているものと一致しなければ $\perp$ を出力する．
  (3) $out = (sid, ek_{ra})$ と分解し，

$$b \leftarrow \text{HW.QuoteVerify}(params_{DE}, quote)$$

  を計算し，$b = 0$ であれば $\perp$ を出力する．
  (4) $state$ から dk と sk を取り出し，

$$ct_{dk} \leftarrow \text{PKE.Enc}(ek_{ra}, dk)$$
$$\sigma_{dk} \leftarrow \text{S.Sign}(sk, (sid, ct_{dk}))$$

  を計算し，$(sid, ct_{dk}, \sigma_{dk})$ を出力する．
- 入力 ("$dec\ keygen$", tag_P) に対し，
  (1) $sk_P = \text{S.Sign}(sk, tag_P)$ を出力する．
- 入力 ("$tokengen$", tag_P, ct_S) に対して，
  (1) $ct_S = (ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m})$ として

$$token_{P,S} \leftarrow \text{S.Sign}(sk, tag_P || H(ct_{i_1} || ct_{i_2} || \cdots || ct_{i_m}))$$

  を出力する．

Experiment $\mathrm{REAL}_{\mathcal{A}}^{\mathrm{ICFE}}(1^\lambda)$:
$(msk, mpk) \leftarrow \mathrm{ICFE.Setup}^{\mathrm{HW}}(1^\lambda)$
$M \leftarrow \mathcal{A}^{\mathrm{FE.DecKeygen}^{\mathrm{HW}}(msk,\cdot),\mathrm{FE.Tokengen}^{\mathrm{HW}}(msk,\cdot,\cdot)}(mpk)$
ただし，$M = \{x_i\}_{i\in[q]}$ である．
$ct_i \leftarrow \mathrm{FE.Enc}(EK, x_i) \quad for\ i \in [n]$
$\alpha \leftarrow \mathcal{A}^{\mathrm{FE.DecKeygen}^{\mathrm{HW}}(msk,\cdot),\mathrm{FE.Tokengen}^{\mathrm{HW}}(msk,\cdot,\cdot),\mathrm{HW}(\cdot),\mathrm{KM}(\cdot)}(ct)$
ここで，$ct = \{ct_i\}_{i\in[q]}$ である．
Output (M, α)

Experiment $\mathrm{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathrm{ICFE}}(1^\lambda)$:
$(mpk) \leftarrow \mathcal{S}(1^\lambda)$
$M \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(mpk)$
ただし，$M = \{x_i\}_{i\in[q]}$ である．
$ct_i \leftarrow \mathcal{S}(1^{|x_i|}) \quad for\ i \in [n]$
$\alpha \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(ct)$
ここで，$ct = \{ct_i\}_{i\in[q]}$ である．
Output (M, α)

図 1　実験 $\mathrm{REAL}_{\mathcal{A}}^{\mathrm{ICFE}}(1^\lambda)$ と実験 $\mathrm{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathrm{ICFE}}(1^\lambda)$
Fig. 1　Experiment $\mathrm{REAL}_{\mathcal{A}}^{\mathrm{ICFE}}(1^\lambda)$ and Experiment $\mathrm{IDEAL}_{\mathcal{A},\mathcal{S}}^{\mathrm{ICFE}}(1^\lambda)$.

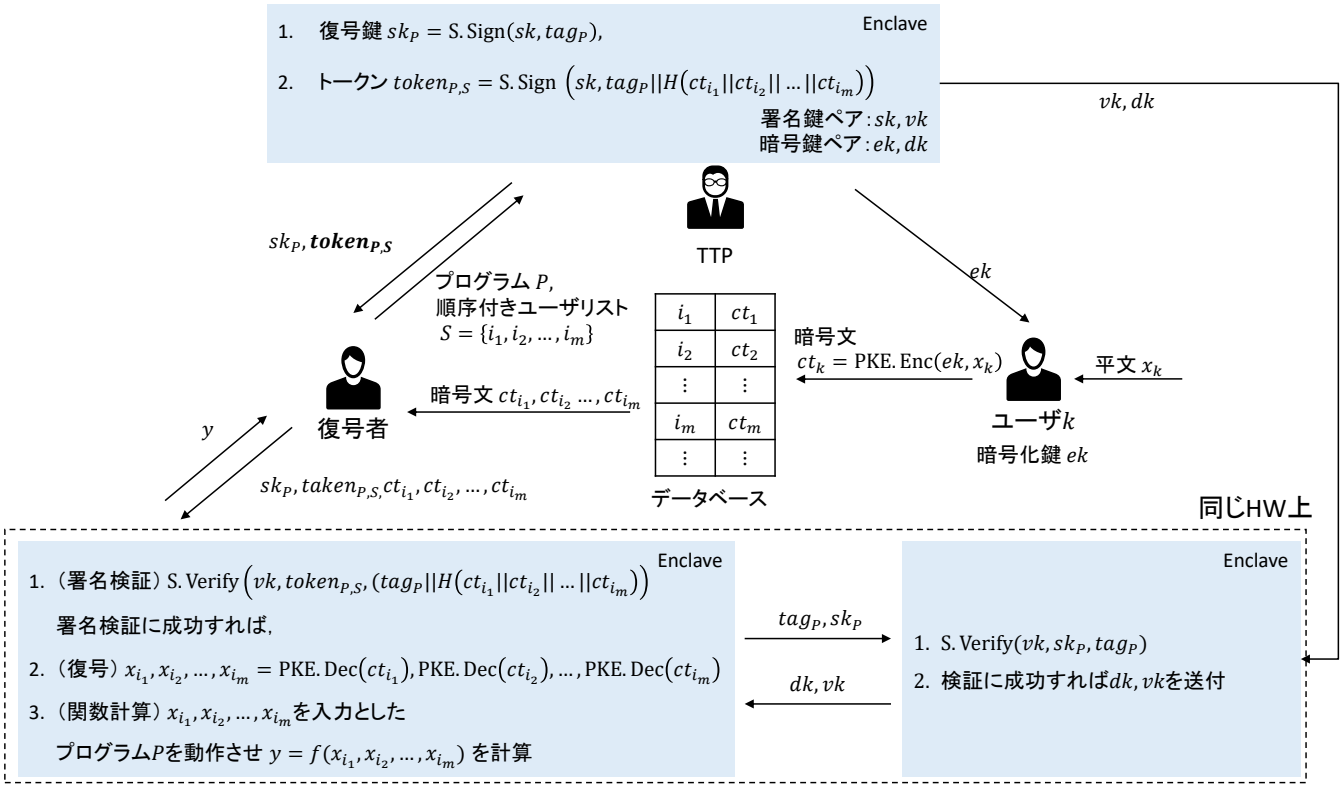

図 2　提案方式の概要
Fig. 2　Overview of the proposed scheme.

Q_{DE}:

- 入力 ("*init setup*", vk) に対して，
  (1) $(ek_{ra}, dk_{ra}) \leftarrow \mathrm{PKE.KeyGen}(1^\lambda)$ を動作させる．
  (2) セッション ID $sid \xleftarrow{r} \{0,1\}^\lambda$ を選ぶ．
  (3) ステート $state$ を (sid, dk_{ra}, vk) に更新し，(sid, ek_{ra}) を出力する．
- 入力 ("*complete setup*", $sid, ct_{dk}, \sigma_{dk}$) に対して，
  (1) $state$ から sid に対するエントリ (sid, dk_{ra}, vk) を取り出す．なければ $\perp$ を出力する．
  (2) $b \leftarrow \mathrm{S.Verify}(vk, \sigma_{dk}, (sid, ct_{dk})$ を計算し，$b = 0$ であれば $\perp$ を出力する．
  (3) $dk \leftarrow \mathrm{PKE.Dec}(dk_{ra}, ct_{dk})$ を計算する．
  (4) (dk, vk) を $state$ に追加する．
- 入力 ("*provision*", $report, sk_P$) に対し，
  (1) $state$ に (dk, vk) がなければ $\perp$ を出力する．
  (2) $state$ に $(report, 1)$ がなければ $\perp$ を出力する．
  (3) $report = (md_{hdl}, tag_Q, in, out, mac)$ と分解し，$b \leftarrow \mathrm{S.Verify}(vk, sk_P, tag_Q)$ を計算する．$b = 0$ であれば $\perp$ を出力する．
  (4) $out = (sid, ek_{la})$ と分解し，

$$ct_{dk}^{la} \leftarrow \mathrm{PKE.Enc}(ek_{la}, dk)$$

  を計算し，(sid, ct_{dk}, vk) を出力する．

$Q_{\mathrm{FE}}(P)$:

- 入力 ("*init setup*") に対して，
  (1) $(ek_{la}, dk_{la}) \leftarrow \mathrm{PKE.KeyGen}(1^\lambda)$ を動作させる．
  (2) セッション ID $sid \xleftarrow{r} \{0,1\}^\lambda$ を選ぶ．
  (3) ステート $state$ を (sid, dk_{la}) に更新し，(sid, ek_{la}) を出力する．
- 入力 ("*complete setup*", $report_{dk}$) に対して，

(1) ステート $state$ に $(report_{dk}, 1)$ がなければ $\perp$ を出力する．

(2) $report_{dk} = (md_{hdl_Q}, tag_Q, in, out, mac)$ と分解し，$out = (sid, ct_{dk}, vk)$ と分解する．

(3) $state$ から sid に対するエントリ (sid, dk_{la}) を取り出す．なければ $\perp$ を出力する．

(4) $dk \leftarrow \mathrm{PKE.Dec}(dk_{la}, ct_{dk})$ を計算する．

(5) $in = (\text{"}provision\text{"}, report, sk_P)$，$report = (md_{hdl_P}, tag_P, in', out', mac'))$ と分解する．

(6) (dk, vk, tag_P) を $state$ に加える．

- 入力 $(\text{"}run\text{"}, token_{P,S}, ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m})$ に対して，

(1) $state$ からエントリ (dk, vk, tag_P) を取り出し，

$$b \leftarrow \mathrm{S.Verify}(vk, token_{P,S}, tag_P || H(ct_{i_1} || ct_{i_2} || \cdots || ct_{i_m}))$$

を計算する．$b = 0$ ならば $\perp$ を出力する．

(2) すべての $k \in [m]$ に対して，$x_{i_k} \leftarrow \mathrm{PKE.Dec}(dk, ct_{i_k})$ を計算する．

(3) $y = P(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ を計算する．

提案する入力制御可能な関数型暗号の各アルゴリズムは以下の通りである．

$\mathrm{ICFE.Setup}^{\mathrm{HW}}(1^\lambda)$:

$$hdl_{KME} \leftarrow \mathrm{HW.Load}(params, Q_{KME}),$$
$$(ek, vk) \leftarrow \mathrm{HW.Run}(hdl_{KME}, (\text{"}init\text{"}, 1^\lambda))$$

を動作させ，$msk = hdl_{KME}$，$mpk = (ek, vk)$ を出力する．

$\mathrm{ICFE.Enc}(mpk, x_k)$:

$mpk = (ek, vk)$ として $ct_k \leftarrow \mathrm{PKE.Enc}(ek, x_k)$ を計算し，ct_k を出力する．

$\mathrm{ICFE.DecKeygen}^{\mathrm{HW}}(msk, P)$:

$msk = hdl_{KME}$ とする．tag_P を計算し，

$$sk_P \leftarrow \mathrm{HW.Run}(hdl_{KME}, (\text{"}dec\ keygen\text{"}, tag_P))$$

を動作させ，sk_P を出力する．

$\mathrm{ICFE.Tokengen}^{\mathrm{HW}}(msk, P, S)$:

$msk = hdl_{KME}$ とする．$S = \{i_1, i_2, \ldots, i_m\}$ に対応する暗号文のリスト $(ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m})$ を取得する．tag_P を計算し，

$$token_{P,S} \leftarrow \mathrm{HW.Run}\ (hdl_{KME}, (\text{"}tokengen\text{"}, tag_P, ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m}))$$

を動作させ，$token_{P,S}$ を出力する．

$\mathrm{ICFE.DecSetup}^{\mathrm{KM,HW}}(mpk)$:

$hdl_{DE} \leftarrow \mathrm{HW.Load}(params, Q_{DE})$ を動作させる．$mpk = (ek, vk)$ として，

$$quote \leftarrow \mathrm{HW.Run\&Quote}(hdl_{DE}, (\text{"}init\ setup\text{"}), vk)$$

を動作させ，$\mathrm{KM}(quote)$ を呼ぶ．実際には，

$$(sid, ct_{dk}, \sigma_{dk}) \leftarrow \mathrm{HW.Run}(hdl_{KME}, (\text{"}provision\text{"}, quote, parames))$$

が実行される．

$$\mathrm{HW.Run}(hdl_{DE}, (\text{"}complete\ setup\text{"}, sid, ct_{dk}, \sigma_{dk}))$$

を動作させ，hdl_{DE} を出力する．

$\mathrm{ICFE.Dec}^{\mathrm{HW}}(hdl, sk_P, token_{P,S}, ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m})$:

$hdl = hdl_{DE}$ とする．

$$hdl_P \leftarrow \mathrm{HW.Load}(params, Q_{FE}(P))$$

を動作させ，

$$report_{dk} \leftarrow \mathrm{HW.Run\&Report}(hdl_P, \text{"}init\ setup\text{"})$$

を呼ぶ．

$$\mathrm{HW.ReportVerify}(hdl_{DE}, report)$$

を動作させ，

$$report_{dk} \leftarrow \mathrm{HW.Run\&Report}(hdl_{DE}, (\text{"}provision\text{"}, report, sk_P))$$

を呼ぶ．

$$\mathrm{HW.ReportVerify}(hdl_P, report_{dk})$$

を動作させ，

$$\mathrm{HW.Run}(hdl_P, (\text{"}complete\ setup\text{"}, report_{dk}))$$

を呼ぶ．最後に，

$$y \leftarrow \mathrm{HW.Run}(hdl_P, (\text{"}run\text{"}, token_{P,S}, ct_{i_1}, ct_{i_2}, \ldots, ct_{i_m}))$$

を実行し，y を出力する．

定理 1. PKE が IND-CCA2 安全性，S が EUF-CMA 安全，H が衝突困難，HW が Report 偽造不可能性および Quote 偽造不可能性を満たすとき，提案方式は任意の q に対して q-シミュレーション安全性を満たす．

証明は吉野らの方式の証明 [5] と同様である．

参考文献

[1] Boneh, D., Sahai, A. and Waters, B.: Functional encryption, *Communications of the ACM*, Vol. 55, No. 11, p. 56 (online), DOI: 10.1145/2366316.2366333 (2012).

[2] Fisch, B., Vinayagamurthy, D., Boneh, D. and Gorbunov, S.: IRON: Functional Encryption using Intel SGX, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, ACM

Press, (online), DOI: 10.1145/3133956.3134106 (2017).
[3] Goldwasser, S., Gordon, S. D., Goyal, V., Jain, A., Katz, J., Liu, F.-H., Sahai, A., Shi, E. and Zhou, H.-S.: Multi-input Functional Encryption, *Advances in Cryptology – EUROCRYPT 2014*, Springer Berlin Heidelberg, pp. 578–602 (online), DOI: 10.1007/978-3-642-55220-5_32 (2014).
[4] Intel Corporation: Intel Software Guard Extensions, `https://software.intel.com/en-us/sgx`.
[5] 吉野慎司，手塚真徹，品川和雅，田中圭介：Intel SGX を用いた入力を制御できる複数入力関数型暗号，2019 年暗号と情報セキュリティシンポジウム (SCIS2019) (2019).