Verifiable Functional Encryption using Intel SGX*

Tatsuya Suzuki[1], Keita Emura[2], Toshihiro Ohigashi[3, 2], and Kazumasa Omote[1, 2]

[1]University of Tsukuba, Japan.
[2]National Institute of Information and Communications Technology (NICT), Japan.
[3]Tokai University, Japan.

September 8, 2021

Abstract

Most functional encryption schemes implicitly assume that inputs to decryption algorithms, i.e., secret keys and ciphertexts, are generated honestly. However, they may be tampered by malicious adversaries. Thus, verifiable functional encryption (VFE) was proposed by Badrinarayanan et al. in ASIACRYPT 2016 where anyone can publicly check the validity of secret keys and ciphertexts. They employed indistinguishability-based (IND-based) security due to an impossibility result of simulation-based (SIM-based) VFE even though SIM-based security is more desirable. In this paper, we propose a SIM-based VFE scheme. To bypass the impossibility result, we introduce a trusted setup assumption. Although it appears to be a strong assumption, we demonstrate that it is reasonable in a hardware-based construction, e.g., Fisch et al. in ACM CCS 2017. Our construction is based on a verifiable public-key encryption scheme (Nieto et al. in SCN 2012), a signature scheme, and a secure hardware scheme, which we refer to as VFE-HW. Finally, we discuss an implementation of VFE-HW using Intel Software Guard Extensions (Intel SGX).

1 Introduction

Functional Encryption: Cloud computing has gained increasing attention since it supports several functionalities, e.g., data analysis. However, sensitive user data must be secured, and protected. Thus, since Public-Key Encryption (PKE) only provides all-or-nothing decryption capabilities, functional encryption [14] has been proposed. Functional encryption allows clients to flexibly access sensitive data toward usual "all or nothing" decryption procedure. Briefly, a Trusted Authority (TA) first generates a master public key mpk and a master secret key msk. A client sends the information of function P to the TA. Generally, P can enforce sophisticated functions, e.g., access control etc. The TA generates a secret key sk_P using the msk, and gives it to the client. A plaintext msg is encrypted by the mpk, where CT is the ciphertext. Finally, the client obtains $\mathsf{P}(\mathsf{msg})$ by decrypting CT using sk_P.

The security of functional encryption is defined by indistinguishability-based (IND-based) or simulation-based (SIM-based) notions. IND-based security guarantees that no adversary can distinguish which plaintext was encrypted. IND-based functional encryption schemes have been proposed

*The main part of this work was done when the first author, Tatsuya Suzuki, was a master student at the Tokai University, Japan, and was a research assistant at the National Institute of Information and Communications Technology (NICT), Japan. The first author is supported by a JSPS Fellowship for Young Scientists. An extended abstract appeared at the 15th International Conference on Provable and Practical Security, ProvSec 2021.

Table 1: Comparison of Verifiable Functional Encryption

	Security	Functionality	Verifiability	Secure HW	Trusted Setup
Fisch et al. [24] (Functional Encryption)	SIM-based	Any	Not Considered	Yes	Yes [1]
Badrinarayanan et al. [10]	IND-based	Limited	Normal	No	No
Soroush et al. [36]	IND-based	Limited	Normal	No	No
Our VFE scheme	SIM-based	Any	Weak	Yes	Yes

for the class of all (polynomial-sized) functionalities under inefficient assumptions, e.g., multi-linear maps, or indistinguishability obfuscation [15, 26, 27, 38]. Consequently, Abdalla et al. [2] proposed an IND-based functional encryption scheme that supports inner products under simple assumptions, and several works followed this direction [3, 4, 19, 22, 23, 37]. However, Boneh et al. [14] and O'Neil [34] demonstrated that IND-based functional encryption yields insufficient security. For example, an adversary is allowed to obtain secret keys for a function P selected by the adversary with the restriction $\mathsf{P}(\mathsf{msg}_0^*) = \mathsf{P}(\mathsf{msg}_1^*)$ where msg_0^* and msg_1^* are challenge plaintexts with the condition $\mathsf{msg}_0^* \neq \mathsf{msg}_1^*$. Thus, the class of P remains restricted, e.g., we cannot specify a cryptographic hash fuction as P due to collision resistance. Thus, SIM-based security is more desirable. Several SIM-based functional encryption schemes [6–8, 14, 18, 34] have been proposed recently. However, several works [6, 7, 14, 18] have shown that achieving SIM-based functional encryption that supports all (polynomial-sized) functionalities is impossible.

Functional Encryption using Intel SGX: To overcome this impossibility result, Fisch et al. [24] proposed IRON, a SIM-based functional encryption scheme that uses Intel SGX [9, 30–32]. Intel SGX is a hardware protection set that protects sensitive data (e.g. medical data) from malicious adversaries by storing them in enclaves generated as isolated spaces in an application. They employed a secure hardware scheme (HW) which modeled Intel SGX.

Briefly, IRON is described as follows. The TA generates a public key pk and a decryption key dk for a PKE scheme, as well as a verification key vk and a signing key sk for a signature scheme (SIG). Then, the TA generates a secret key sk_P, where P is a function for the client. The TA generates a signature of P as a secret key sk_P using sk in a Key Manager Enclave (KME), and sends it to the client. Let CT be the ciphertext of a plaintext msg under pk. In the decryption procedure, if sk_P is a valid signature using vk, CT is decrypted inside an enclave, and $\mathsf{P}(\mathsf{msg})$ is output.

Verifiable Functional Encryption: Most functional encryption schemes implicitly assume that inputs to decryption algorithm, i.e., sk_P and CT, are generated honestly according to the algorithmic procedures. However, they may be tampered by malicious adversaries. Badrinarayanan et al. [10] proposed Verifiable Functional Encryption (VFE). With VFE, anyone can publicly check the validity of sk_P and CT. If verification of sk_P and CT passes, the decryption algorithm of VFE correctly outputs $\mathsf{P}(\mathsf{msg})$. Badrinarayanan et al. insisted that VFE are useful for some applications, e.g., storing encrypted images [14] and audits [29]. As a drawback, they demonstrated that SIM-based VFE implies the existence of one message zero-knowledge proof systems for **NP** in the plain model. This implication contradicts the impossibility result shown by Goldreich et al. [28]. We emphasize that IRON does not help us to bypass this impossibility result. As a result, they employed IND-based security as shown in Table 1. A VFE proposed by Soroush et al. [36], which supports inner products, employs the same IND-based security definition. Thus, no SIM-based VFE has been proposed so far.

[1] The $\mathsf{HW.Setup}$ algorithm in the pre-processing phase is required to be honestly run by the TA.

Our Contribution: We propose a SIM-based VFE scheme that supports any (polynomial-sized) functionality. To support such functionality, we employ the hardware-based construction given in IRON [24], and, to achieve SIM-based security, we relax the verifiability of the definition given by Badrinarayanan et al. without losing the practicability. Intuitively, we assume that mpk and msk are generated honestly whereas those can be arbitrary values in the definition given by Badrinarayanan et al. Due to this trusted setup assumption, mpk can be considered a common reference string (CRS) in the one message zero-knowledge context [13]. One may think that this trusted setup assumption is unreasonable and too strong in practice. However, this is not the case in the hardware-based construction. We will explain it in detail in Section 4.

In addition to provide a security definition that bypasses the impossibility result, we also give a SIM-based VFE construction. The original IRON has supported public verifiability of secret keys (because these are signatures), thus we focus on how to support public verifiability for ciphertexts. Therefore, we employ (publicly) Verifiable PKE (VPKE) [33] proposed by Nieto et al. in addition to the ingredients of IRON (PKE, SIG, and HW). We employ HW as in IRON, thus we refer to proposed system as $\mathsf{VFE\text{-}HW}$. Note that publicly executable computations should be run outside of memory-constrained enclaves as much as possible. Simultaneously, as in IRON, ciphertexts input to enclaves require to be non-malleable, and thus the underlying $\mathsf{(V)PKE}$ scheme needs to be CCA-secure. Consequently, we modify the definition of VPKE (Section 2).

Finally, we give our implementation of the proposed $\mathsf{VFE\text{-}HW}$ scheme for a cryptographic hash function H as the function P, i.e., the decryption algorithm for a ciphertext of msg outputs $\mathsf{H(msg)}$. Due to the nonlinearity of the hash function, the functionality seems hard to be supported by functional encryption with linear computations, e.g., inner products. Moreover, the IND-based VFE scheme does not support the function due to the key generation query restriction. In addition to these theoretical perspectives, it seems meaningful to support this functionality in practice, e.g., a password PW is encrypted and $\mathsf{H(PW)}$ can be computed without revealing PW. Here, we employ the Pairing-Based Cryptography (PBC) library [1] to implement the VPKE scheme proposed by Nieto et al. Briefly, the encryption algorithm runs in 0.11845 sec, the verification algorithm for ciphertexts runs in 0.12329 sec, the verification algorithm for secret keys runs in 0.00057 sec, and the decryption algorithm runs in 0.06164 sec. This is an extended abstract appeared at the 15th International Conference on Provable and Practical Security, ProvSec 2021.

2 Preliminaries

Here, we define PKE, VPKE, SIG, and HW. When x is selected uniformly from set S, we denote this as $x \xleftarrow{\$} S$, and $y \leftarrow A(x)$ represents that y is the output of an algorithm A with an input x.

First, we introduce the definition of PKE as follows. Let $\mathcal{M}_{\mathsf{pke}}$ be a plaintext space of PKE.

Definition 1 (Syntax of PKE). *A PKE scheme* PKE *consists of the following three algorithms,* $\mathsf{PKE.KeyGen}$*,* $\mathsf{PKE.Enc}$*, and* $\mathsf{PKE.Dec}$*:*

$\mathsf{PKE.KeyGen}(1^\lambda)$: *This key generation algorithm takes as input the security parameter* $\lambda \in \mathbb{N}$*, and return a public key* $\mathsf{pk_{pke}}$ *and a secret key* $\mathsf{dk_{pke}}$*.*

$\mathsf{PKE.Enc}(\mathsf{pk_{pke}}, \mathsf{msg})$: *This encryption algorithm takes as input* $\mathsf{pk_{pke}}$*, a plaintext* $\mathsf{msg} \in \mathcal{M}_{\mathsf{pke}}$*, and returns a ciphertext* CT*.*

$\mathsf{PKE.Dec}(\mathsf{dk_{pke}}, \mathsf{CT})$: *This decryption algorithm takes as input* $\mathsf{dk_{pke}}$*, and* CT*, and returns a plaintext* msg *or reject symbol* $\perp$*.*

Correctness is defined as follows: For all $(\mathsf{pk_{pke}}, \mathsf{dk_{pke}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$, all $\mathsf{msg} \in \mathcal{M}_{\mathsf{pke}}$, and $\mathsf{PKE.Dec}(\mathsf{dk_{pke}}, \mathsf{CT}) = \mathsf{msg}$ holds, where $\mathsf{CT} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk_{pke}}, \mathsf{msg})$.

Next, we define indistinguishability against chosen ciphertext attack (IND-CCA) as follows.

Definition 2 (IND-CCA). *For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ and the security parameter $\lambda \in \mathbb{N}$, we define the experiment $\mathrm{Exp}^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{A}}(\lambda)$ as follows. Here,* state *is state information that an adversary $\mathcal{A}$ can preserve any information, and* state *is used for transferring state information to the other stage.*

$$\begin{aligned}
&\mathrm{Exp}^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{A}}(\lambda):\\
&\quad (\mathsf{pk_{pke}}, \mathsf{dk_{pke}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)\\
&\quad (\mathsf{msg}_0^*, \mathsf{msg}_1^*, \mathsf{state}) \leftarrow \mathcal{A}^{\mathsf{PKE.DEC}}(\mathsf{find}, \mathsf{pk_{pke}})\\
&\quad \mathsf{msg}_0^*, \mathsf{msg}_1^* \in \mathcal{M}_{\mathsf{pke}};\ |\mathsf{msg}_0^*| = |\mathsf{msg}_1^*|\\
&\quad \mu \xleftarrow{\$} \{0,1\};\ \mathsf{CT}^* \leftarrow \mathsf{PKE.Enc}(\mathsf{pk_{pke}}, \mathsf{msg}_\mu^*)\\
&\quad \mu' \leftarrow \mathcal{A}^{\mathsf{PKE.DEC}}(\mathsf{guess}, \mathsf{CT}^*, \mathsf{state})\\
&\quad \textit{If } \mu = \mu' \textit{ then output } 1,\ \textit{and } 0 \textit{ otherwise}
\end{aligned}$$

- $\mathsf{PKE.DEC}$*: This decryption oracle takes as input a ciphertext* $\mathsf{CT} \neq \mathsf{CT}^*$ *and returns* msg *by running the* $\mathsf{PKE.Dec}(\mathsf{dk_{pke}}, \mathsf{CT})$ *algorithm.*

We say that PKE *is IND-CCA secure if the advantage*

$$\mathrm{Adv}^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{A}}(\lambda) :=\mid \Pr[\mathrm{Exp}^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{A}}(\lambda) = 1] - 1/2 \mid$$

is negligible for any PPT adversary $\mathcal{A}$.

Next, we introduce the definition of SIG as follows. Let $\mathcal{M}_{\mathsf{sig}}$ be a message space.

Definition 3 (Syntax of SIG). *A signature scheme* SIG *consists of the following three algorithms,* $\mathsf{SIG.KeyGen}$*,* $\mathsf{SIG.Sign}$ *and* $\mathsf{SIG.Verify}$*:*

$\mathsf{SIG.KeyGen}(1^\lambda)$: *This key generation algorithm takes as input the security parameter $\lambda \in \mathbb{N}$, and returns a signing/verification key pair* $(\mathsf{sk_{sign}}, \mathsf{vk_{sign}})$.

$\mathsf{SIG.Sign}(\mathsf{sk_{sign}}, \mathsf{msg})$: *This signing algorithm takes as input* $\mathsf{sk_{sign}}$ *and a message* $\mathsf{msg} \in \mathcal{M}_{\mathsf{sig}}$*, and returns a signature σ.*

$\mathsf{SIG.Verify}(\mathsf{vk_{sign}}, \mathsf{msg}, \sigma)$: *This verification algorithm takes as input* $\mathsf{vk_{sign}}, \mathsf{msg}$ *and σ, and returns* 1 *(valid) or* 0 *(invalid).*

Correctness is defined as follows: For all $(\mathsf{sk_{sign}}, \mathsf{vk_{sign}}) \leftarrow \mathsf{SIG.KeyGen}(1^\lambda)$ and all $\mathsf{msg} \in \mathcal{M}_{\mathsf{sig}}$, $\mathsf{SIG.Verify}(\mathsf{vk_{sign}}, \mathsf{msg}, \sigma) = 1$ holds, where $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{sk_{sign}}, \mathsf{msg})$.

Next, we define existential unforgeability against chosen message attack (EUF-CMA) of SIG as follows.

Definition 4 (EUF-CMA). *For any PPT adversary $\mathcal{A}$ and the security parameter $\lambda \in \mathbb{N}$, we define the experiment* $\mathrm{Exp}^{\mathrm{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(\lambda)$ *as follows.*

$$
\begin{aligned}
&\mathrm{Exp}^{\mathrm{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(1^\lambda):\\
&\quad (\mathsf{sk_{sign}}, \mathsf{vk_{sign}}) \leftarrow \mathsf{SIG.KeyGen}(1^\lambda);\ \ \mathsf{QUERY} := \emptyset\\
&\quad (\mathsf{msg}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{SIG.SIGN}}(\mathsf{vk_{sign}})\\
&\quad \textit{If } \mathsf{SIG.Verify}(\mathsf{vk_{sign}}, \mathsf{msg}^*, \sigma^*) = 1 \textit{ and } \mathsf{msg}^* \notin \mathsf{QUERY}\\
&\quad\ \textit{then output } 1,\ \textit{and } 0 \textit{ otherwise}
\end{aligned}
$$

- $\mathsf{SIG.SIGN}$*: This signing oracle takes as input a message* msg*, and returns* σ *by running the* $\mathsf{SIG.Sign}(\mathsf{sk_{sign}}, \mathsf{msg})$ *algorithm. Finally, the challenger stores* msg *in* QUERY*.*

We say that SIG *is EUF-CMA secure if the advantage*

$$\mathrm{Adv}^{\mathrm{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(\lambda) := \Pr[\mathrm{Exp}^{\mathrm{EUF\text{-}CMA}}_{\mathsf{SIG},\mathcal{A}}(\lambda) = 1]$$

is negligible for any PPT adversary $\mathcal{A}$*.*

Next, we introduce VPKE as defined by Nieto et al. [33]. VPKE provides public verifiability, where anyone can check the validity of ciphertexts without using any secret value. They defined the decryption algorithm $\mathsf{VPKE.Dec}$ using two algorithms, i.e., the verification algorithm $\mathsf{VPKE.Ver}$ and the decryption algorithm for converted ciphertext $\mathsf{VPKE.Dec}'$. $\mathsf{VPKE.Ver}$ verifies ciphertext CT and converts CT to CT' if CT is valid. $\mathsf{VPKE.Dec}'$ decrypts CT', and outputs msg. In this paper, we further decompose $\mathsf{VPKE.Ver}$ into two algorithms, i.e., $\mathsf{VPKE.Ver}$ and $\mathsf{VPKE.Conv}$, which will be explained later. The verification algorithm $\mathsf{VPKE.Ver}$ verifies CT and the conversion algorithm $\mathsf{VPKE.Conv}$ converts CT into CT'.

Next, we define VPKE. Here, let $\mathcal{M}_{\mathsf{vpke}}$ be a plaintext space of VPKE.

Definition 5 (Syntax of VPKE).

$\mathsf{VPKE.PGen}(1^\lambda)$: *This public parameter generation algorithm takes the security parameter* $\lambda \in \mathbb{N}$ *as input, and returns a public parameter* pars*.*

$\mathsf{VPKE.KeyGen}(\mathsf{pars})$: *This key generation algorithm takes* pars *as input, and returns a public key* $\mathsf{pk_{vpke}}$ *and a secret key* $\mathsf{dk_{vpke}}$*.*

$\mathsf{VPKE.Enc}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{msg})$: *This encryption algorithm takes* pars*,* $\mathsf{pk_{vpke}}$ *and a plaintext* $\mathsf{msg} \in \mathcal{M}_{\mathsf{vpke}}$ *as input, and returns a ciphertext* CT*.*

$\mathsf{VPKE.Dec}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}, \mathsf{CT})$: *This decryption algorithm takes* pars*,* $\mathsf{pk_{vpke}}$*,* $\mathsf{dk_{vpke}}$ *and* CT *as input, and returns a plaintext* msg *or reject symbol* $\perp$*. Internally the algorithm runs* $\mathsf{VPKE.Ver}$*,* $\mathsf{VPKE.Conv}$*, and* $\mathsf{VPKE.Dec}'$*, which are defined as follows.*

$\mathsf{VPKE.Ver}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT})$: *This verification algorithm takes* pars*,* $\mathsf{pk_{vpke}}$ *and* CT *as input, and returns 1 or 0.*

$\mathsf{VPKE.Conv}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT})$: *This conversion algorithm takes* pars*,* $\mathsf{pk_{vpke}}$ *and* CT *as input, and returns a ciphertext* CT'*.*

$\mathsf{VPKE.Dec}'(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}, \mathsf{CT}')$: *This decryption algorithm takes* pars*,* $\mathsf{pk_{vpke}}$*,* $\mathsf{dk_{vpke}}$ *and* CT' *as input, and returns a plaintext* msg*.*

Correctness is defined as follows: For all $\mathsf{pars} \leftarrow \mathsf{VPKE.PGen}(1^\lambda)$, all $(\mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}) \leftarrow \mathsf{VPKE.KeyGen}$ (pars), all $\mathsf{msg} \in \mathcal{M}_{\mathsf{vpke}}$, $\mathsf{VPKE.Dec}'(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}$ and $\mathsf{VPKE.Conv}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT})) = \mathsf{msg}$ holds, where $\mathsf{CT} \leftarrow \mathsf{VPKE.Enc}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{msg})$ and $\mathsf{VPKE.Ver}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT}) = 1$.

Next, we define strictly non-trivial public verification. Condition 1 requires that the decryption of a ciphertext CT succeeds if and only if its verification outputs 1, and Condition 2 excludes CCA-secure schemes where the decryption algorithm does not output $\perp$.

Definition 6 (Strictly Non-Trivial Public Verification). *For any PPT adversary $\mathcal{A}$ and the security parameter $\lambda \in \mathbb{N}$, let $\mathsf{pars} \leftarrow \mathsf{VPKE.PGen}(1^\lambda)$. We define the $\mathsf{VPKE.Ver}$ algorithm is strictly non-trivial public verifiable if* (1) $(\mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}) \leftarrow \mathsf{VPKE.KeyGen}(\mathsf{pars})$, *and* $\mathsf{VPKE.Ver}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT}) = 0 \iff \mathsf{VPKE.Dec}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}, \mathsf{CT}) = \perp$ *for all* CT, *and* (2) *there exists a ciphertext* CT *for which* $\mathsf{VPKE.Dec}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}, \mathsf{CT}) = \perp$ *are provided.*

Next, we define IND-CCA as follows.

Definition 7 (IND-CCA). *For any PPT adversary $\mathcal{A}$ and the security parameter $\lambda \in \mathbb{N}$, we define the experiment $\mathrm{Exp}^{\text{IND-CCA}}_{\mathsf{VPKE},\mathcal{A}}(\lambda)$ as follows. Here,* state *is state information that an adversary $\mathcal{A}$ can preserve any information, and* state *is used for transferring state information to the other stage.*

$$
\begin{aligned}
&\mathrm{Exp}^{\text{IND-CCA}}_{\mathsf{VPKE},\mathcal{A}}(\lambda):\\
&\quad \mathsf{pars} \leftarrow \mathsf{VPKE.PGen}(1^\lambda);\ (\mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}) \leftarrow \mathsf{VPKE.KeyGen}(\mathsf{pars})\\
&\quad (\mathsf{msg}_0^*, \mathsf{msg}_1^*, \mathsf{state}) \leftarrow \mathcal{A}^{\mathsf{VPKE.DEC}}(\mathsf{find}, \mathsf{pars}, \mathsf{pk_{vpke}})\\
&\quad \mathsf{msg}_0^*, \mathsf{msg}_1^* \in \mathcal{M}_{\mathsf{vpke}};\ |\mathsf{msg}_0^*| = |\mathsf{msg}_1^*|\\
&\quad \mu \xleftarrow{\$} \{0,1\};\ \mathsf{CT}^* \leftarrow \mathsf{VPKE.Enc}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{msg}_\mu^*)\\
&\quad \mu' \leftarrow \mathcal{A}^{\mathsf{VPKE.DEC}}(\mathsf{guess}, \mathsf{CT}^*, \mathsf{state})\\
&\quad \textit{If } \mu = \mu' \textit{ then output } 1,\ \textit{and } 0 \textit{ otherwise}
\end{aligned}
$$

- $\mathsf{VPKE.DEC}$: *This decryption oracle takes a ciphertext* $\mathsf{CT} \neq \mathsf{CT}^*$ *as input. If* $\mathsf{VPKE.Ver}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT}) = 0$, *output* $\perp$. *Otherwise, compute* $\mathsf{CT}' \leftarrow \mathsf{VPKE.Conv}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT})$, *and return* msg *by running the* $\mathsf{VPKE.Dec}'(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}, \mathsf{CT}')$ *algorithm.*

We say that VPKE *is IND-CCA secure if the advantage* $\mathrm{Adv}^{\text{IND-CCA}}_{\mathsf{VPKE},\mathcal{A}}(\lambda) := |\Pr[\mathrm{Exp}^{\text{IND-CCA}}_{\mathsf{VPKE},\mathcal{A}}(\lambda) = 1] - 1/2|$ *is negligible for any PPT adversary $\mathcal{A}$.*

For the sake of clarity, we give the Nieto et al. VPKE scheme employed in our implementation in the Appendix A.

Next, we define the secure hardware scheme (HW scheme) [24]. In this paper, the hardware instance HW denotes an oracle that provides the functionalities given in Definition 8. Furthermore, the hardware oracle $\mathsf{HW}(\cdot)$ denotes an interaction with other local secure hardware in addition to HW, and the Key Manager oracle $\mathsf{KM}(\cdot)$ denotes an interaction with a remote secure hardware over an untrusted channel.

Definition 8 (Syntax of HW Scheme). *A* HW *scheme for a set of probabilistic programs $\mathcal{Q}$ comprises the following seven algorithms.* HW *has variables* $\mathsf{HW.sk_{report}}$, $\mathsf{HW.sk_{quote}}$, *and a table* T. *Here,* $\mathsf{HW.sk_{report}}$ *and* $\mathsf{HW.sk_{quote}}$ *are leveraged to store keys, and the table* T *is leveraged to manage the internal state of loaded enclave programs.*

- $\mathsf{HW.Setup}(1^\lambda)$: *This hardware setup algorithm takes the security parameter* $\lambda \in \mathbb{N}$ *as input, and returns a public parameters* params*. This algorithm also generates the secret keys* $\mathsf{sk_{report}}$ *and* $\mathrm{sk}_{\mathrm{quote}}$*, and stores these keys in the* $\mathsf{HW.sk_{report}}$ *and* $\mathsf{HW.sk_{quote}}$ *valuables respectively.*

- $\mathsf{HW.Load}(\mathsf{params}, \mathsf{Q})$: *This loading program algorithm takes* params *and a program* $\mathsf{Q} \in \mathcal{Q}$ *as input, and returns a handle* hdl*. Intuitively, this algorithm loads the stateful program into the enclave to be launched. Here,* hdl *is leveraged to identify the enclave running* Q*.*

- $\mathsf{HW.Run}(\mathsf{hdl}, \mathsf{in})$: *This running program algorithm takes* hdl *and a symbol* in *as input, and returns* out *corresponding to an enclave running a designated program* Q*. Intuitively, this algorithm runs* Q *at state* $\mathsf{T}[\mathsf{hdl}]$ *with* in*, and records* out*.*

- $\mathsf{HW.Run\&Report}_{\mathsf{sk_{report}}}(\mathsf{hdl}, \mathsf{in})$: *This running program and generating report algorithm, which can be verified by an enclave program on the same hardware platform for a local attestation, takes* hdl *and* in *as input, and returns a report* $\mathsf{report} := (\mathsf{md_{hdl}}, \mathsf{tag_Q}, \mathsf{in}, \mathsf{out}, \mathsf{mac})$*, where* $\mathsf{md_{hdl}}$ *is a metadata relative enclave,* $\mathsf{tag_Q}$ *is a program tag that identifies the program running inside an enclave, and* mac *is a message authentication code produced using* $\mathsf{sk_{report}}$ *for* $(\mathsf{md_{hdl}}, \mathsf{tag_Q}, \mathsf{in}, \mathsf{out})$*.*

- $\mathsf{HW.Run\&Quote}_{\mathsf{sk_{quote}}}(\mathsf{hdl}, \mathsf{in})$: *This running program and generating quote algorithm, which can be publicly verified different hardware platform for a remote attestation, takes* hdl *and* in *as input, and returns a quote* $\mathsf{quote} := (\mathsf{md_{hdl}}, \mathsf{tag_Q}, \mathsf{in}, \mathsf{out}, \sigma)$*, where* $\mathsf{md_{hdl}}$ *is a metadata relative enclave,* $\mathsf{tag_Q}$ *is a program tag that identifies the program running inside an enclave, and* σ *is a signature produced using* $\mathsf{sk_{quote}}$ *for* $(\mathsf{md_{hdl}}, \mathsf{tag_Q}, \mathsf{in}, \mathsf{out})$*.*

- $\mathsf{HW.ReportVerify}_{\mathsf{sk_{report}}}(\mathsf{hdl}, \mathsf{report})$: *This report verification algorithm takes* hdl *and* report *as input, and uses* $\mathsf{sk}_{\mathrm{report}}$ *to verify* mac*. If* mac *is valid, then the algorithm outputs* 1 *and adds a tuple* $(1, \mathsf{report})$ *to* $\mathsf{T}[\mathsf{hdl}]$*. Otherwise, the algorithm outputs* 0 *and adds tuple* $(0, \mathsf{report})$ *to* $\mathsf{T}[\mathsf{hdl}]$*.*

- $\mathsf{HW.QuoteVerify}(\mathsf{params}, \mathsf{quote})$: *This quote verification algorithm, takes* params *and* quote *as input. This algorithm verifies* σ*. If the verification of* σ *succeeds, then the algorithm outputs* 1*. Otherwise,* 0 *is output.*

Correctness is defined as follows: HW is correct if the following things hold. For all $\mathsf{Q} \in \mathcal{Q}$, in in the input domain of Q and all handles hdl

- Correctness of Run: $\mathsf{out} = \mathsf{Q}(\mathsf{in})$ if Q is deterministic. More generally, $\exists$ random coins r (sampled in time and used by Q) such that $\mathsf{out} = \mathsf{Q}(\mathsf{in})$.

- Correctness of Report and $\mathsf{ReportVerify}$: $\Pr[\mathsf{HW.Report\text{-}Verify}_{\mathrm{sk}_{\mathrm{report}}}(\mathsf{hdl}, \mathsf{report}) = 0] = \mathrm{negl}(\lambda)$

- Correctness of Quote and $\mathsf{QuoteVerify}$: $\Pr[\mathsf{HW.Quote\text{-}Verify}(\mathsf{params}, \mathsf{quote}) = 0] = \mathrm{negl}(\lambda)$

Next, we define local attestation unforgeability (LOC-ATT-UNF) of HW as follows. This security guarantees that no adversary that does not have $\mathsf{sk_{report}}$ can produce a valid report.

Definition 9 (LOC-ATT-UNF) *For any PPT adcersary* $\mathcal{A}$ *and the security parameter* $\lambda \in \mathbb{N}$*, we define the experiment* $\mathrm{Exp}^{\text{LOC-ATT-UNF}}_{\mathsf{HW},\mathcal{A}}(\lambda)$ *as follows.*

$$
\begin{aligned}
&\mathrm{Exp}_{\mathsf{HW},\mathcal{A}}^{\text{LOC-ATT-UNF}}(\lambda):\\
&\quad (\mathsf{params}, \mathsf{sk}_{\mathsf{report}}, \mathsf{sk}_{\mathsf{quote}}, \mathsf{state}) \leftarrow \mathsf{HW.Setup}(1^\lambda)\\
&\quad \mathsf{QUERY} := \emptyset \ ; (\mathsf{hdl}^*, \mathsf{report}^*) \leftarrow \mathcal{A}^{\mathsf{HW},\mathsf{HW}(\cdot)}(\mathsf{params})\\
&\quad \textit{If } \mathsf{HW.ReportVerify}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}^*, \mathsf{report}^*) = 1 \textit{ where}\\
&\quad \mathsf{report}^* = (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_{\mathsf{Q}}, \mathsf{in}^*, \mathsf{out}^*, \mathsf{mac}^*) \textit{ and}\\
&\quad (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_{\mathsf{Q}}, \mathsf{in}^*, \mathsf{out}^*) \ \notin \ \mathsf{QUERY}\\
&\quad \textit{then output } 1, \ \textit{and } 0 \textit{ otherwise}
\end{aligned}
$$

- HW*: $\mathcal{A}$ can access the instance as follows.*
  - $\mathsf{HW.LOAD}$*: $\mathcal{A}$ queries the instance as input* params *and* Q*, and the instance returns the handle* hdl *by running the* $\mathsf{HW.Load}(\mathsf{params}, \mathsf{Q})$ *algorithm.*
  - $\mathsf{HW.REPORTVERIFY}$*: $\mathcal{A}$ queries the instance as input* hdl *and* report*, and the instance returns the result by running the* $\mathsf{HW.ReportVerify}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}, \mathsf{report})$ algorithm.
- $\mathsf{HW}(\cdot)$*: $\mathcal{A}$ can access the oracle as follows.*
  - $\mathsf{HW.RUN\&REPORT}$ *: $\mathcal{A}$ queries the oracle as input* hdl *and* in*, and the oracle returns* $\mathsf{report} := (\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{\mathsf{Q}}, \mathsf{in}, \mathsf{out}, \mathsf{mac})$ *by running the* $\mathsf{HW.Run\&Report}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}, \mathsf{in})$ *algorithm. Finally, the oracle stores* $(\mathsf{md}_{\mathsf{hdl}}, \mathsf{tag}_{\mathsf{Q}}, \mathsf{in}, \mathsf{out})$ *in* QUERY*.*

We say that HW *is LOC-ATT-UNF secure if the advantage*

$$\mathrm{Adv}_{\mathsf{HW},\mathcal{A}}^{\text{LOC-ATT-UNF}}(\lambda) := \Pr[\mathrm{Exp}_{\mathsf{HW},\mathcal{A}}^{\text{LOC-ATT-UNF}}(\lambda) = 1]$$

is negligible for any PPT adversary $\mathcal{A}$.

Next, we define remote attestation unforgeability (REM-ATT-UNF) of HW as follows. This security guarantees that no adversary that does not have $\mathsf{sk}_{\mathsf{quote}}$ can produce a valid quote.

Definition 10 (REM-ATT-UNF) *For any PPT adcersary $\mathcal{A}$ and the security parameter $\lambda \in \mathbb{N}$, we define the experiment* $\mathrm{Exp}_{\mathsf{HW},\mathcal{A}}^{\text{REM-ATT-UNF}}(\lambda)$ *as follows.*

$$
\begin{aligned}
&\mathrm{Exp}_{\mathsf{HW},\mathcal{A}}^{\text{REM-ATT-UNF}}(\lambda):\\
&\quad (\mathsf{params}, \mathsf{sk}_{\mathsf{report}}, \mathsf{sk}_{\mathsf{quote}}, \mathsf{state}) \leftarrow \mathsf{HW.Setup}(1^\lambda)\\
&\quad \mathsf{QUERY} := \emptyset \ ; \mathsf{quote}^* \leftarrow \mathcal{A}^{\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{params})\\
&\quad \textit{If } \mathsf{HW.QuoteVerify}(\mathsf{params}, \mathrm{quote}) = 1 \textit{ where}\\
&\quad \mathsf{quote}^* = (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_{\mathsf{Q}}, \mathsf{in}^*, \mathsf{out}^*, \sigma) \textit{ and}\\
&\quad (\mathsf{md}^*_{\mathsf{hdl}}, \mathsf{tag}^*_{\mathsf{Q}}, \mathsf{in}^*, \mathsf{out}^*) \ \notin \ \mathsf{QUERY}\\
&\quad \textit{then output } 1, \ \textit{and } 0 \textit{ otherwise}
\end{aligned}
$$

- HW*: $\mathcal{A}$ can access the instance as follows.*
  - $\mathsf{HW.LOAD}$*: $\mathcal{A}$ queries the instance as input* params *and* Q*, and the instance returns the handle* hdl *by running the* $\mathsf{HW.Load}(\mathsf{params}, \mathsf{Q})$ *algorithm.*

- $\mathsf{KM}(\cdot)$: *$\mathcal{A}$ can access the oracle as follows.*
  - HW.RUN&QUOTE: *$\mathcal{A}$ queries the oracle as input* hdl *and* in, *and the oracle returns* $\mathsf{quote} := (\mathsf{md_{hdl}}, \mathsf{tag_Q}, \mathsf{in}, \mathsf{out}, \sigma)$ *by running the* $\mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathrm{quote}}}(\mathsf{hdl}, \mathsf{in})$ *algorithm. Finally, the oracle stores* $(\mathsf{md_{hdl}}, \mathsf{tag_Q}, \mathsf{in}, \mathsf{out})$ *in* QUERY.

We say that HW *is REM-ATT-UNF secure if the advantage*

$$\mathrm{Adv}^{\text{REM-ATT-UNF}}_{\mathsf{HW},\mathcal{A}}(\lambda) := \Pr[\mathrm{Exp}^{\text{REM-ATT-UNF}}_{\mathsf{HW},\mathcal{A}}(\lambda) = 1]$$

is negligible for any PPT adversary $\mathcal{A}$.

3 Impossibility Result of VFE and Our Solution

In this section, we recall the impossibility result of VFE shown by Badrinarayanan et al. [10]. We remark that this impossibility is caused by the verifiability of VFE. Thus, they have mentioned that even if the impossibility of SIM-based security given by Agrawal et al. [6] is bypassed, still the impossibility of VFE remains.

Since their VFE syntax is differ from our VFE-HW, first we introduce their syntax as follows. The setup algorithm $\mathsf{VFE.Setup}(1^\lambda)$ generates (mpk, msk), the key-generation algorithm $\mathsf{VFE.KeyGen}(\mathsf{mpk}, \mathsf{msk}, \mathsf{P})$ outputs $\mathsf{sk_P}$, the encryption algorithm $\mathsf{VFE.Enc}(\mathsf{mpk}, \mathsf{msg})$ outputs CT, and the decryption algorithm $\mathsf{VFE.Dec}(\mathsf{mpk}, \mathsf{P}, \mathsf{sk_P}, \mathsf{CT})$ outputs $\mathsf{P}(\mathsf{msg})$ or $\perp$. In addition to these algorithms, VFE supports two verification algorithms. The ciphertext verification algorithm $\mathsf{VFE.VerifyCT}(\mathsf{mpk}, \mathsf{CT})$ outputs 0 or 1, and the secret key verification algorithm $\mathsf{VFE.VerifyK}(\mathsf{mpk}, \mathsf{P}, \mathsf{sk_P})$ outputs 0 or 1.

Next, we introduce verifiability defined by them as follows. The verifiability guarantees that if ciphertexts and secret keys are verified by the respective algorithms then each ciphertext should be associated with a unique message msg, and the decryption result is $\mathsf{P}(\mathsf{msg})$. We remark that it holds even under possibly maliciously generated mpk. Let $\mathcal{P}_{\mathsf{VFE}}$ and $\mathcal{M}_{\mathsf{VFE}}$ be a family of function for VFE and a plaintext space of VFE respectively.

Definition 11 (Verifiability). *For all security parameter $\lambda \in \mathbb{N}$, $\mathsf{mpk} \in \{0,1\}^*$, and all $\mathsf{CT} \in \{0,1\}^*$, there exists $\mathsf{msg} \in \mathcal{M}_{\mathsf{VFE}}$ such that for all $\mathsf{P} \in \mathcal{P}_{\mathsf{VFE}}$ and $\mathsf{sk_P} \in \{0,1\}^*$, if $\mathsf{VFE.VerifyCT}(\mathsf{mpk}, \mathsf{CT}) = 1$ and $\mathsf{VFE.VerifyK}(\mathsf{mpk}, \mathsf{P}, \mathsf{sk_P}) = 1$, then $\Pr[\mathsf{VFE.Dec}(\mathsf{mpk}, \mathsf{P}, \mathsf{sk_P}, \mathsf{CT}) = \mathsf{P}(\mathsf{msg})] = 1$ holds.*

We further remark that the probability that the $\mathsf{VFE.Dec}$ algorithm outputs $\mathsf{P}(\mathsf{msg})$ is exactly 1 if CT and $\mathsf{sk_P}$ are valid. Thus, Badrinarayanan et al. assumed that perfect correctness holds (otherwise, a non-uniform malicious authority can sample ciphertexts/keys from the space where it fails to be correct). We note that the probability is exactly 1 yields perfect soundness for all adversaries when a proof system is constructed from VFE.

Next, we describe the impossibility result as follows.

Theorem 1 ([10], Theorem 3) *There exists a family of functions, each of which can be represented as a polynomial sized circuit, for which there does not exist any simulation secure verifiable functional encryption scheme.*

To prove the theorem, Badrinarayanan et al. showed that SIM-based VFE implies the existence of one message zero-knowledge proof system for **NP** in the plain model which is known to be impossible. More concretely, let L be a **NP** complete language and R be the relation of L which takes as input a string x and a polynomial sized (in the length of x) witness ω. $\mathsf{R}(x, \omega)$ outputs 1 if and only if $x \in L$ and ω is its witness. We denote $\mathsf{R}(x, \cdot)$ for all $x \in \{0,1\}^\lambda$. A one message zero-knowledge proof system $(\mathcal{P}, \mathcal{V})$ for the language L with relation R is constructed from VFE as follows. For (x, ω), the prover $\mathcal{P}$ runs $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{VFE.Setup}(1^\lambda)$ where $\lambda = |x|$, computes $\mathsf{CT} \leftarrow \mathsf{VFE.Enc}(\mathrm{mpk}, \omega)$ and $\mathsf{sk}_\mathsf{R}(x, \cdot) \leftarrow \mathsf{VFE.KeyGen}(\mathsf{mpk}, \mathsf{msk}, \mathsf{R}(x, \cdot))$, and outputs a proof $\pi = (\mathsf{mpk}, \mathsf{CT}, \mathsf{sk}_\mathsf{R}(x, \cdot))$. The verifier $\mathcal{V}$ accepts π if $\mathsf{VFE.Dec}(\mathsf{mpk}, \mathsf{R}(x, \cdot), \mathsf{sk}_\mathsf{R}(x, \cdot), \mathsf{CT}) = 1$. Obviously, the proof system is perfectly complete if the underlying VFE scheme is perfectly correct. Moreover, due to the verifiability property, the system is perfectly sound. Furthermore, since the verifiability holds even for maliciously generated mpk, CT, and sk, no trusted setup is assumed. Due to the SIM-based security, i.e., the existence of the simulator that can produce a ciphertext only from $\mathsf{R}(x, \omega)$ without knowing ω (here, $1 = \mathsf{R}(x, \omega)$ in this case), the system provides computational zero knowledge.

To bypass the impossibility result, we introduce the trusted setup where $(\mathsf{mpk}, \mathsf{msk})$ is generated honestly, and mpk is considered as a CRS. [2] One may think that this trusted setup assumption is unreasonable and too strong in practice. However, this is not the case in the hardware-based construction. In our system, mpk and msk are generated by running a setup program, and it is implicitly assumed that the setup program is executed correctly (Q in our scheme). That is, anyone can verify the description of the function. Moreover, we assume that the program is hardcoded as the static data, and is assumed to be not tampered. The remaining is to trust the computer that correctly runs the program, and is widely assumed when cryptographic protocols are implemented. Thus, we claim that the trusted assumption is reasonable, and leave how to remove the assumption without losing the SIM-based security as a future work of this paper.

We remark that even if one message zero-knowledge proof system in the CRS model can be constructed from SIM-based VFE, this does not bypass the impossibility result since the proof system in the plain model implies a proof system in the CRS model. We emphasize that the setup algorithm that generates $(\mathsf{mpk}, \mathsf{msk})$ must be run first since other algorithms take mpk or msk as input. Due to this situation, we can bypass the impossibility result of Badrinarayanan et al. since any VFE-based one message zero-knowledge proof system or argument need to run the Setup algorithm first, and then mpk can be seen as a CRS. As mentioned by Barak and Pass [11], one message zero-knowledge proofs and arguments can be constructed in the CRS model (without certain relaxations).

Regarding the CRS model, Badrinarayanan et al. have mentioned that VFE seems to be constructed from a functional encryption scheme with Non-Interactive Zero-Knowledge (NIZK) proof systems. However, the CRS may be maliciously generated and then soundness does not hold. Thus, they gave up for employing NIZK proof systems and employed non-interactive witness indistinguishable proof (NIWI) systems as the ingredients. Since we introduce the trusted setup assumption, we may be able to construct VFE from this direction without employing a HW scheme. However, even then, another impossibility arises [6]. For bypassing the impossibility, we employ a HW scheme.

Random oracles may be employed to avoid introducing the trusted setup assumption. However, as mentioned by Agrawal, Koppula, and Waters [7], there is an impossibility result of SIM-based

[2] We note that we also relax the condition that the verifiability holds where the probability that the decryption algorithm outputs $\mathsf{P}(\mathsf{msg})$ is not exactly 1 (concretely $1 - \mathsf{negl}(\lambda)$) in our definition. Because the underlying local or remote attestations require non-perfect correctness, this relaxation is reasonable. This relaxation provides the converted proof system to be an argument, i.e., soundness holds only for computationally bounded adversaries.

security in the random oracle model. Thus, we do not further consider the random oracle model in this paper.

4 Definitions of VFE-HW

In this section, we define $\mathsf{VFE\text{-}HW}$. Here, let HW be a hardware instance that takes a handle hdl that identifies an enclave. If an algorithm is allowed to access HW, then the algorithm can use the secure hardware functionality given in Definition 8. Let $\mathsf{HW}(\cdot)$ (resp. $\mathsf{KM}(\cdot)$) be a hardware (resp. a key manager) oracle that takes hdl and an authentication information (Report (resp. Quote) in our construction), interacts with other local enclave specified by hdl, and runs the function contained in the authentication information. Let $\mathcal{P}_{\mathsf{VFE\text{-}HW}}$ and $\mathcal{M}_{\mathsf{VFE\text{-}HW}}$ be a family of functions for $\mathsf{VFE\text{-}HW}$ and a plaintext space of $\mathsf{VFE\text{-}HW}$ respectively.

Definition 12 (Syntax of $\mathsf{VFE\text{-}HW}$). *A* $\mathsf{VFE\text{-}HW}$ *scheme comprises the following seven algorithms:*

$\mathsf{VFE\text{-}HW.Setup}^{\mathsf{HW}}(1^\lambda)$: *This setup algorithm takes the security parameter* $\lambda \in \mathbb{N}$ *as input, and returns a master public key* mpk *and a master secret key* msk.

$\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}(\mathsf{msk}, \mathsf{P})$: *This key generation algorithm takes* msk *and a function* $\mathsf{P} \in \mathcal{P}_{\mathsf{VFE\text{-}HW}}$ *as input, and returns a secret key* $\mathsf{sk_P}$ *for* P.

$\mathsf{VFE\text{-}HW.Enc}(\mathsf{mpk}, \mathsf{msg})$: *This encryption algorithm takes* mpk *and a plaintext* $\mathsf{msg} \in \mathcal{M}_{\mathsf{VFE\text{-}HW}}$ *as input, and returns a ciphertext* CT.

$\mathsf{VFE\text{-}HW.DecSetup}^{\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{mpk})$: *This decryption node setup algorithm takes* mpk *as input, and returns a handle* hdl.

$\mathsf{VFE\text{-}HW.VerifyCT}(\mathsf{mpk}, \mathsf{CT})$: *This ciphertext verification algorithm takes* mpk *and* CT *as input, and returns* 1 *or* 0.

$\mathsf{VFE\text{-}HW.VerifyK}(\mathsf{mpk}, \mathsf{P}, \mathsf{sk_P})$: *This secret key verification algorithm takes* mpk, P*, and* $\mathsf{sk_P}$ *as input, and returns* 1 *or* 0.

$\mathsf{VFE\text{-}HW.Dec}^{\mathsf{HW}(\cdot)}(\mathsf{mpk}, \mathsf{hdl}, \mathsf{P}, \mathsf{sk_P}, \mathsf{CT})$: *This decryption algorithm takes* $\mathsf{mpk}, \mathsf{hdl}, \mathsf{sk_P}$*, and* CT *as input, and returns a value* $\mathsf{P}(\mathsf{msg})$ *or a reject symbol* $\perp$.

Correctness is defined as follows: For all $\mathsf{P} \in \mathcal{P}_{\mathsf{VFE\text{-}HW}}$, all $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{VFE\text{-}HW.Setup}^{\mathsf{HW}}(1^\lambda)$, all $\mathsf{sk_P} \leftarrow \mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}(\mathsf{msk}, \mathsf{P})$, all $\mathsf{hdl} \leftarrow \mathsf{VFE\text{-}HW.DecSetup}^{\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{mpk})$, and all $\mathsf{msg} \in \mathcal{M}_{\mathsf{VFE\text{-}HW}}$, let $\mathsf{CT} \leftarrow \mathsf{VFE\text{-}HW.Enc}(\mathsf{mpk}, \mathsf{msg})$, then $\Pr[\mathsf{VFE\text{-}HW.Dec}^{\mathsf{HW}(\cdot)}(\mathsf{mpk}, \mathsf{hdl}, \mathsf{sk_P}, \mathsf{CT}) = \mathsf{P}(\mathsf{msg})] = 1 - \mathsf{negl}(\lambda)$ holds.

Next we define weak verifiability. As mentioned in Section 3, we somewhat relax the original verifiability definition, i.e., we employ the trusted setup and the probability of verifiability is not exactly 1 due to the correctness of HW scheme. Thus, we call our definition weak verifiability. Weak verifiability guarantees that if ciphertexts and secret keys are verified by the respective algorithms, then each ciphertext should be associated with a unique message msg, and the decryption result is $\mathsf{P}(\mathsf{msg})$. Note that this holds only when mpk is generated honestly and hdl is non-$\perp$.

Definition 13 (Weak Verifiability). *For all security parameters* $\lambda \in \mathbb{N}$*,* $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{VFE\text{-}HW.Setup}^{\mathsf{HW}}(1^\lambda)$*, and* $\mathsf{hdl} \leftarrow \mathsf{VFE\text{-}HW.DecSetup}^{\mathsf{HW},\mathsf{KM}(\cdot)}(\mathsf{mpk})$ *where* $\mathsf{hdl} \neq \perp$*, and all* $\mathsf{CT} \in \{0,1\}^*$*, there exists* $\mathsf{msg} \in \mathcal{M}_{\mathsf{VFE\text{-}HW}}$ *such that for all* $\mathsf{P} \in \mathcal{P}_{\mathsf{VFE\text{-}HW}}$ *and* $\mathsf{sk_P} \in \{0,1\}^*$*, if* $\mathsf{VFE\text{-}HW.VerifyCT}(\mathsf{mpk}, \mathsf{CT}) = 1$ *and* $\mathsf{VFE\text{-}HW.VerifyK}(\mathsf{mpk}, \mathsf{P}, \mathsf{sk_P}) = 1$*, then* $\Pr[\mathsf{VFE\text{-}HW.Dec}^{\mathsf{HW}(\cdot)}(\mathsf{mpk}, \mathsf{hdl}, \mathsf{P}, \mathsf{sk_P}, \mathsf{CT}) = \mathsf{P}(\mathsf{msg})] = 1 - \mathsf{negl}(\lambda)$ *holds.*

Next we define the simulation security of VFE-HW as follows. This security guarantees that no adversary can distinguish REAL and IDEAL, where REAL represents the actual environment. Note that msk and the challenge plaintext msg^* are not explicitly used in IDEAL.

Definition 14 (Simulation security). *For a stateful PPT adversary $\mathcal{A}$, a stateful PPT simulator $\mathcal{S}$ and the security parameter $\lambda \in \mathbb{N}$, we define the real experiment $\mathrm{Exp}^{\mathrm{REAL}}_{\mathsf{VFE\text{-}HW}}(\lambda)$ and the ideal experiment $\mathrm{Exp}^{\mathrm{IDEAL}}_{\mathsf{VFE\text{-}HW}}(\lambda)$ as follows. Here, let $\mathsf{U}_{\mathsf{msg}}(\cdot)$ denote a universal oracle where $\mathsf{U}_{\mathsf{msg}}(P) = \mathsf{P}(\mathsf{msg})$.*

$\mathrm{Exp}^{\mathrm{REAL}}_{\mathsf{VFE\text{-}HW}}(\lambda)$:

$(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{VFE\text{-}HW.Setup}^{\mathsf{HW}}(1^\lambda);\ \mathsf{msg}^* \leftarrow \mathcal{A}^{\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}(\mathsf{msk},\cdot)}(\mathsf{mpk})$

$\mathsf{CT}^* \leftarrow \mathsf{VFE\text{-}HW.Enc}(\mathsf{mpk}, \mathsf{msg}^*);\ \alpha \leftarrow \mathcal{A}^{\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}(\mathsf{msk},\cdot),\mathsf{HW}(\cdot),\mathsf{KM}(\cdot)}(\mathsf{mpk}, \mathsf{CT}^*)$

Output (msg^*, α)

- HW: *$\mathcal{A}$ can access the instance as follows.*
  - HW.LOAD: *$\mathcal{A}$ queries the instance as input* params *and* Q, *and the instance returns* hdl *by running the* HW.Load(params, Q) *algorithm.*
  - HW.RUN: *$\mathcal{A}$ queries the instance as input* hdl *and* in, *and the instance returns* out *by running the* HW.Run(hdl, in) *algorithm.*
- $\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}$: *$\mathcal{A}$ queries this key generation oracle as input* msk *and* P. *The oracle accesses* HW.RUN *as input* hdl = msk *and* in = P, *and the oracle returns* $\mathsf{sk_P}$ *as* out *by running the* HW.Run(hdl, in) *algorithm.*
- $\mathsf{HW}(\cdot)$: *$\mathcal{A}$ can access* HW.RUN&REPORT *in addition to* HW *as input* hdl *and* in, *and the oracle returns* report *by running the* $\mathsf{HW.Run\&Report}_{\mathsf{sk}_{\mathrm{report}}}(\mathsf{hdl}, \mathsf{in})$ *algorithm.*
- $\mathsf{KM}(\cdot)$: *$\mathcal{A}$ can access* HW.RUN&QUOTE *as input* hdl *and* in, *and the oracle returns* quote *by running the* $\mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathrm{quote}}}(\mathsf{hdl}, \mathsf{in})$ *algorithm.*

$\mathrm{Exp}^{\mathrm{IDEAL}}_{\mathsf{VFE\text{-}HW}}(\lambda)$:

$\mathsf{mpk} \leftarrow \mathcal{S}(1^\lambda);\ \mathsf{msg}^* \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(\mathsf{mpk})$

$\mathsf{CT}^* \leftarrow \mathcal{S}^{\mathsf{U}_{\mathsf{msg}}(\cdot)}(1^\lambda, 1^{|\mathsf{msg}^*|});\ \alpha \leftarrow \mathcal{A}^{\mathcal{S}^{\mathsf{U}_{\mathsf{msg}}(\cdot)}(\cdot)}(\mathsf{mpk}, \mathsf{CT}^*)$

Output (msg^*, α)

- $\mathcal{S}(\cdot)$: *$\mathcal{S}$ simulates the* HW, $\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}$, $\mathsf{HW}(\cdot)$ *and* $\mathsf{KM}(\cdot)$ *oracles.*
- $\mathcal{S}^{\mathsf{U}_{\mathsf{msg}}(\cdot)}(\cdot)$: *$\mathcal{S}$ simulates the* HW, *the* $\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}$, *the* $\mathsf{HW}(\cdot)$ *and the* $\mathsf{KM}(\cdot)$ *oracles. Here, if $\mathcal{A}$ queries this oracle as input* CT^* *and* $\mathsf{sk_P}$, *$\mathcal{S}$ outputs* P(msg) *using the universal oracle* $\mathsf{U}_{\mathsf{msg}}(\cdot)$ *that inputs* P *queried in the* $\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}$ *oracle.*

If there exists a stateful simulator $\mathcal{S}$ and $\mathrm{Exp}^{\mathrm{REAL}}_{\mathsf{VFE\text{-}HW}}(\lambda)$ and $\mathrm{Exp}^{\mathrm{IDEAL}}_{\mathsf{VFE\text{-}HW}}(\lambda)$ are computationally indistinguishable, then we say that the VFE-HW scheme is simulation secure against a stateful PPT adversary

5 Proposed Scheme

In this section, we describe the proposed $\mathsf{VFE\text{-}HW}$ scheme. The proposed scheme is constructed from IND-CCA secure and strictly non-trivial public verifiable VPKE, IND-CCA secure PKE, EUF-CMA secure SIG and REM-ATT-UNF, and LOC-ATT-UNF secure HW.

High-Level Description: Essencially, we follow the construction of IRON. IRON has supported public verifiability of secret keys (since these are signatures), we focus on supporting the public verifiability of ciphertexts. Therefore, we replace a PKE scheme employed to encrypt msg with a VPKE scheme.

In our $\mathsf{VFE\text{-}HW}$ scheme, the (function) enclave securely executes computations that require secret values, however, its computational power and memory are constrained. Thus, the verification part should be run outside of the enclave, and we employ the public verifiability of VFE. However, the ciphertext is converted if the original $\mathsf{VPKE.Ver}$ algorithm is employed. Thus, the converted ciphertext CT' is decrypted via $\mathsf{VPKE.Dec}'$ in the enclave. Although at least IND-CPA security is guaranteed if $\mathsf{VPKE.Dec}$ is replaced with $\mathsf{VPKE.Dec}'$ [33], the underlying VPKE scheme is required to be CCA-secure. Thus, we decompose $\mathsf{VPKE.Ver}$ to $\mathsf{VPKE.Ver}$ and $\mathsf{VPKE.Conv}$, and run $\mathsf{VPKE.Conv}$ inside of the enclave.

We consider the following assumptions in the construction of the $\mathsf{VFE\text{-}HW}$. The first two assumptions are the same as those of IRON, and we introduce the last assumption in this paper.

- Pre-Processing: The TA and a client need to complete the pre-processing phase before using $\mathsf{VFE\text{-}HW}$ scheme. In our construction, we consider that a manufacturer setups and initializes the secure hardware. A public parameter is generated by this phase independent of the $\mathsf{VFE\text{-}HW}$ algorithms, and this parameter is implicitly given to all algorithms.

- Non-Interaction: In $\mathsf{VFE\text{-}HW}$, a plaintext is encrypted using a public key of a VPKE scheme, and thus the decryption of the ciphertext requires the corresponding decryption key, which differs from a secret key $\mathsf{sk_P}$. To obtain the decryption key from the KME, we require a one-time hardware setup operation. The $\mathsf{VFE\text{-}HW.DecSetup}^{\mathsf{HW},\mathsf{KM}(\cdot)}$ algorithm interacts with the KME via the $\mathsf{KM}(\cdot)$, and the $\mathsf{VFE\text{-}HW.Dec}^{\mathsf{HW}(\cdot)}$ algorithm is non-interactive.

- Trusted Setup: $\mathsf{VFE\text{-}HW.Setup}^{\mathsf{HW}}$ and $\mathsf{VFE\text{-}HW.DecSetup}^{\mathsf{HW},\mathsf{KM}(\cdot)}$ are executed honestly. In short, mpk, msk and hdl are generated honestly.

The proposed scheme is given as follows. First, we describe the programs $\mathsf{Q_{KME}}$ (for the KME), $\mathsf{Q_{DE}}$ (for a Decryption Enclave DE) and $\mathsf{Q_{FE}}$ (for a Function Enclave FE). $\mathsf{Q_{FE}}$ is parameterized by a function P, and thus we denote $\mathsf{Q_{FE}(P)}$. Let T be an internal state valuable, $\mathsf{tag}_{\mathsf{Q_{DE}}}$ be a measurement of $\mathsf{Q_{DE}}$ hardcoded in the static data of $\mathsf{Q_{KME}}$, and $\mathsf{tag}_{\mathsf{Q_{FE}(P)}}$ be a measurement of $\mathsf{Q_{FE}(P)}$.

$\mathsf{Q_{KME}}$:

- On input ("init", 1^λ):

  1. Run $\mathsf{pars} \leftarrow \mathsf{VPKE.PGen}(1^\lambda)$.
  2. Run $(\mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}) \leftarrow \mathsf{VPKE.KeyGen}(\mathsf{pars})$ and $(\mathsf{sk_{sign}}, \mathsf{vk_{sign}}) \leftarrow \mathsf{SIG.KeyGen}(1^\lambda)$.
  3. Update T to $(\mathsf{dk_{vpke}}, \mathsf{sk_{sign}}, \mathsf{vk_{sign}})$ and output $(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{vk_{sign}})$.

- On input ("$\mathsf{provision}$", $\mathsf{quote}, \mathsf{params}$):

  1. Parse $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl_{DE}}}, \mathsf{tag}_{\mathsf{Q_{DE}}}, \mathsf{in}, \mathsf{out}, \sigma)$. If $\mathsf{tag}_{\mathsf{Q_{DE}}}$ is not matched to tag hardcoded as static data, then output $\perp$.

2. Parse $\mathsf{in} = (\text{"}\mathsf{init\ setup}\text{"}, \mathsf{vk_{sign}})$ and check if $\mathsf{vk_{sign}}$ matches with one in T.
3. Parse $\mathsf{out} = (\mathsf{sid}, \mathsf{pk_{ra}})$ and run $\mathsf{b} \leftarrow \mathsf{HW.QuoteVerify}(\mathsf{params}, \text{quote})$. If $\mathsf{b} = 0$ output $\perp$.
4. Retrieve $\mathsf{dk_{vpke}}$ from T and compute $\mathsf{ct_{dk}} = \mathsf{PKE.Enc}(\mathsf{pk_{ra}}, \mathsf{dk_{vpke}})$ and $\sigma_{\mathsf{dk}} = \mathsf{SIG.Sign}(\mathsf{sk_{sign}}, (\mathsf{sid}, \mathsf{dk_{vpke}}))$, and output $(\mathsf{sid}, \mathsf{ct_{dk}}, \sigma_{\mathsf{dk}})$.

- On input $(\text{"}\mathsf{sign}\text{"}, \mathsf{msg})$: Compute $\mathsf{sig} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk_{sign}}, \mathsf{msg})$ and output sig.

$\mathsf{Q_{DE}}$:

- On input $(\text{"}\mathsf{init\ setup}\text{"}, \mathsf{vk_{sign}})$:
  1. Run $(\mathsf{pk_{ra}}, \mathsf{dk_{ra}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
  2. Generate a session ID, $\mathsf{sid} \leftarrow \{0,1\}^\lambda$.
  3. Update T to $(\mathsf{sid}, \mathsf{dk_{ra}}, \mathsf{vk_{sign}})$ and output $(\mathsf{sid}, \mathsf{pk_{ra}})$.

- On input $(\text{"}\mathsf{complete\ setup}\text{"}, \mathsf{pk_{ra}}, \mathsf{sid}, \mathsf{ct_{dk}}, \sigma_{\mathsf{dk}})$:
  1. Look up T to obtain the entry $(\mathsf{sid}, \mathsf{dk_{ra}}, \mathsf{vk_{sign}})$. If no entry exists for sid, output $\perp$.
  2. If $\mathsf{SIG.Verify}(\mathsf{vk_{sign}}, (\mathsf{sid}, \mathsf{ct_{dk}}), \sigma_{\mathsf{dk}}) = 0$, output $\perp$. Otherwise, run $\mathsf{dk_{vpke}} \leftarrow \mathsf{PKE.Dec}(\mathsf{dk_{ra}}, \mathsf{ct_{dk}})$.
  3. Add the tuple $(\mathsf{dk_{vpke}}, \mathsf{vk_{sign}})$ to T.

- On input $(\text{"}\mathsf{provision}\text{"}, \mathsf{report}, \mathsf{sig})$:
  1. Check to see that the setup has been completed, i.e. T contains the tuple $(\mathsf{dk_{vpke}}, \mathsf{vk_{sign}})$. If not, output $\perp$.
  2. Check to see that the report has been verified, i.e. T contains the tuple $(1, \mathsf{report})$. If not, output $\perp$.
  3. Parse $\mathsf{report} = (\mathsf{md_{hdl_P}}, \mathsf{tag_{Q_{FE}}}(\mathsf{P}), \mathsf{in}, \mathsf{out}, \mathsf{mac})$ and parse $\mathsf{out} = (\mathsf{sid}, \mathsf{pk_{la}})$.
  4. If $\mathsf{SIG.Verify}(\mathsf{vk_{sign}}, \mathsf{tag_{Q_{FE}(P)}}, \mathsf{sig}) = 0$, then output $\perp$. Otherwise, output $(\mathsf{sid}, \mathsf{ct_{key}} = \mathsf{PKE.Enc}(\mathsf{pk_{la}}, \mathsf{dk_{vpke}}))$.

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

- On input $(\text{"}\mathsf{init}\text{"}, \mathsf{sig})$:
  1. Run $(\mathsf{pk_{la}}, \mathsf{dk_{la}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
  2. Generate a session ID, $\mathsf{sid} \leftarrow \{0,1\}^\lambda$.
  3. Update T to $(\mathsf{sid}, \mathsf{dk_{la}})$ and output $(\mathsf{sid}, \mathsf{pk_{la}})$.

- On input $(\text{"}\mathsf{run}\text{"}, \mathsf{pars}, \mathsf{params}, \mathsf{mpk}, \mathsf{pk_{la}}, \mathsf{report_{dk}}, \mathsf{CT})$:
  1. Parse $\mathsf{mpk} = (\mathsf{pk_{vpke}}, \mathsf{vk_{sign}})$.
  2. Check to see that the report has been verified, i.e. T contains the tuple $(1, \mathsf{report_{dk}})$. If not, output $\perp$.
  3. Parse $\mathsf{report_{dk}} = (\mathsf{md_{hdl_{DE}}}, \mathsf{tag_{Q_{DE}}}, \mathsf{in}, \mathsf{out}, \mathsf{mac})$. Parse $\mathsf{out} = (\mathsf{sid}, \mathsf{ct_{key}})$.
  4. Look up T to obtain the entry $(\mathsf{sid}, \mathsf{dk_{la}}, \mathsf{sk_P})$. If no entry exists for sid, output $\perp$.
  5. Compute $\mathsf{dk_{vpke}} \leftarrow \mathsf{PKE.Dec}(\mathsf{dk_{la}}, \mathsf{ct_{key}})$.
  6. Compute $\mathsf{CT'} \leftarrow \mathsf{VPKE.Conv}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{CT})$.

7. Compute $\mathsf{msg} \leftarrow \mathsf{VPKE.Dec}'(\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{dk}_{\mathsf{vpke}}, \mathsf{CT}')$.
8. Evaluate P on msg using sk_{P} and record the output $\mathsf{out} := \mathsf{P}(\mathsf{msg})$. Output out.

Next, we describe the proposed scheme as follows. Here, without loss of generality, prior to running $\mathsf{VFE\text{-}HW.Dec}$, we assume that a ciphertext CT is verified by $\mathsf{VFE\text{-}HW.VerifyCT}$, and a secret key sk_{P} is verified by $\mathsf{VFE\text{-}HW.VerifyK}$. Then, CT and sk_{P} are input to $\mathsf{VFE\text{-}HW.Dec}$ only when these are valid, and $\mathsf{VFE\text{-}HW.Dec}$ does not check their validity. This assumption is natural because we consider public verifiability for both CT and sk_{P}.

Proposed scheme:

Pre-Processing phase : The trusted authority platform and decryption node run respectively.

1. Call $\mathsf{params} \leftarrow \mathsf{HW.Setup}(1^\lambda)$, and output params.

$\mathsf{VFE\text{-}HW.Setup}^{\mathsf{HW}}(1^\lambda)$:

1. Call $\mathsf{hdl}_{\mathsf{KME}} \leftarrow \mathsf{HW.Load}(\mathsf{params}, \mathsf{Q}_{\mathsf{KME}})$.
2. Call $(\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{vk}_{\mathsf{sign}}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{KME}}, (\text{"init"}, 1^\lambda))$.
3. Output $\mathsf{mpk} = (\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{vk}_{\mathsf{sign}}), \mathsf{msk} = \mathsf{hdl}_{\mathsf{KME}}$.

$\mathsf{VFE\text{-}HW.Keygen}^{\mathsf{HW}}(\mathsf{msk}, \mathsf{P})$:

1. Parse $\mathsf{msk} = \mathsf{hdl}_{\mathsf{KME}}$.
2. Compute $\mathsf{tag}_{\mathsf{P}}$ by using a function P.
3. Call $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{KME}}, (\text{"sign"}, \mathsf{tag}_{\mathsf{P}}))$.
4. Output $\mathsf{sk}_{\mathsf{P}} = \mathsf{sig}$.

$\mathsf{VFE\text{-}HW.Enc}(\mathsf{mpk}, \mathsf{msg})$:

1. Parse $\mathsf{mpk} = (\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{vk}_{\mathsf{sign}})$.
2. Compute $\mathsf{CT} \leftarrow \mathsf{VPKE.Enc}(\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{msg})$.

$\mathsf{VFE\text{-}HW.DecSetup}^{\mathsf{HW}, \mathsf{KM}(\cdot)}(\mathsf{mpk})$:

1. Call $\mathsf{hdl}_{\mathsf{DE}} \leftarrow \mathsf{HW.Load}(\mathsf{params}, \mathsf{Q}_{\mathsf{DE}})$.
2. Parse $\mathsf{mpk} = (\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{vk}_{\mathsf{sign}})$.
3. Call $\mathsf{quote} \leftarrow \mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{quote}}}(\mathsf{hdl}_{\mathsf{DE}}, (\text{"}\mathsf{init\ setup}\text{"}, \mathsf{vk}_{\mathsf{sign}}))$.
4. Call $\mathsf{KM}(\mathsf{quote})$ which internally run $(\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}, \sigma_{\mathsf{dk}}) \leftarrow \mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{KME}}, (\text{"}\mathsf{provision}\text{"}, \mathsf{quote}, \mathsf{params}))$.

$\mathsf{VFE\text{-}HW.VerifyCT}(\mathsf{mpk}, \mathsf{CT})$:

1. Parse $\mathsf{mpk} = (\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{vk}_{\mathsf{sign}})$.
2. If $\mathsf{VPKE.Ver}(\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{CT}) = \perp$, then output $\mathsf{0}$. Otherwise, output $\mathsf{1}$.

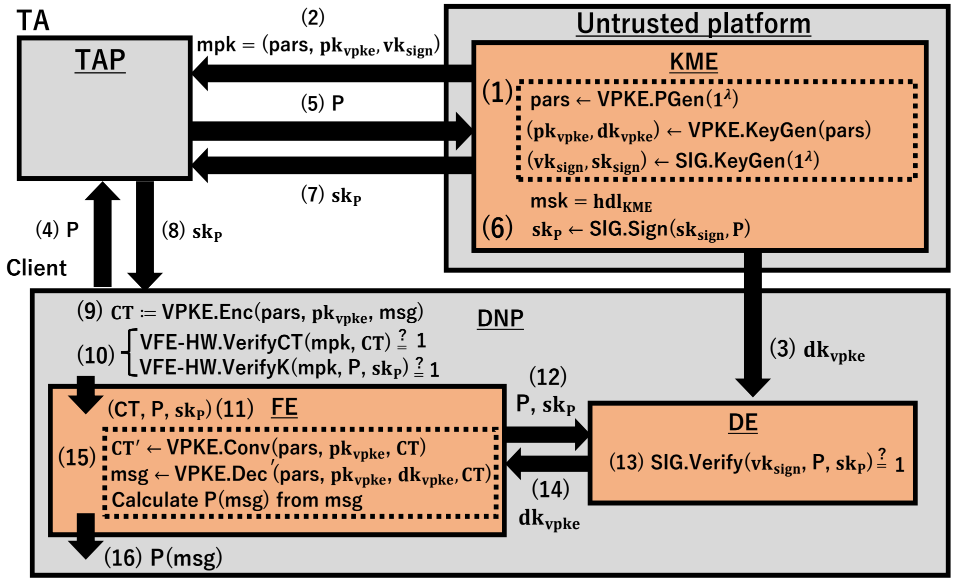

Figure 1: Protocol flow. Steps (1) and (2) specify VFE-HW.Setup, step (3) specifies VFE-HW.DecSetup, steps (4), (5), (6), (7) and (8) specify VFE-HW.KeyGen, step (9) specifies VFE-HW.Enc, steps (10) and (11) specify VFE-HW.VerifyK and VFE-HW.VerifyCT, and steps (12), (13), (14), (15) and (16) specify VFE-HW.Dec.

$\mathsf{VFE\text{-}HW.VerifyK}(\mathsf{mpk}, \mathsf{P}, \mathsf{sk_P})$:

1. Parse $\mathsf{mpk} = (\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{vk_{sign}})$, and $\mathsf{sk_P} = \mathsf{sig}$.
2. If $\mathsf{SIG.Verify}(\mathsf{vk_{sign}}, \mathsf{sk_P}, \mathsf{P}) = 0$, then output 0. Otherwise, output 1.

$\mathsf{VFE\text{-}HW.Dec}^{\mathsf{HW}(\cdot)}(\mathsf{mpk}, \mathsf{hdl}, \mathsf{P}, \mathsf{sk_P}, \mathsf{CT})$:

1. Parse $\mathsf{mpk} = (\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{vk_{sign}})$, $\mathsf{hdl} = \mathsf{hdl_{DE}}$, $\mathsf{sk_P} = \mathsf{sig}$.
2. Call $\mathsf{hdl_{FE}}(\mathsf{P}) \leftarrow \mathsf{HW.Load}(\mathsf{params}, \mathsf{Q_{FE}}(\mathsf{P}))$.
3. Call $\mathsf{report} \leftarrow \mathsf{HW.Run\&Report}_{\mathsf{sk_{report}}}(\mathsf{hdl_{FE}}(\mathsf{P}), (\text{“init”}, \mathsf{sig}))$.
4. If $\mathsf{HW.ReportVerify}_{\mathrm{sk_{report}}}(\mathsf{hdl_{DE}}, \mathsf{report}) = 0$, then output $\perp$. Otherwise, call $\mathsf{report_{dk}} \leftarrow \mathsf{HW.Run\&Report}_{\mathsf{sk_{report}}}(\mathsf{hdl_{DE}}, (\text{“provision”}, \mathsf{report}, \mathsf{sig}))$.
5. If $\mathsf{HW.ReportVerify}_{\mathsf{sk_{report}}}(\mathsf{hdl_{FE}}(\mathsf{P}), \mathsf{report_{dk}}) = 0$, then output $\perp$. Otherwise, call $\mathsf{out} \leftarrow \mathsf{HW.Run}(\mathsf{hdl_P}, (\text{“run”}, \mathsf{pars}, \mathsf{params}, \mathsf{mpk}, \mathsf{pk_{la}}, \mathsf{report_{dk}}, \mathsf{CT}))$, and output out.

Obviously, correctness holds if VPKE, PKE, SIG, and HW are correct.

For clarity, we describe the protocol flow of VFE-HW using Figure 1, where the gray areas represent the untrusted space of each platform, orange areas represent the trusted space of each platform, and the procedures inside dashed boxes are run within enclaves. For example, the TA manages a Trusted Authority Platform TAP, and setups the KME in the TAP. A client manages a Decryption Node Platform (DNP), and setups a DE in the DNP. The TA generates a public key $\mathsf{pk_{vpke}}$ and a secret key $\mathsf{dk_{vpke}}$, as well as a signing key $\mathsf{sk_{sign}}$ and a verification key $\mathsf{vk_{sign}}$ as step (1) within KME. Here, mpk generated by the $\mathsf{VFE\text{-}HW.Setup}^{\mathsf{HW}}$ algorithm consists of pars,

$\mathsf{pk_{vpke}}$ and $\mathsf{vk_{sign}}$ as step (2). Furthermore, msk generated by the $\mathsf{VFE\text{-}HW.Setup^{HW}}$ algorithm is a handle $\mathsf{hdl_{KME}}$ used to confirm the KME. Next, the client preserves $\mathsf{dk_{vpke}}$ into the DE via a remote attestation as step (3). Next, the client gets the secret key $\mathsf{sk_P}$ of the $\mathsf{VFE\text{-}HW.KeyGen^{HW}}$ algorithm which KME issues as a signature on a function P via a secure channel as step (4) to (8). Here, let CT be a ciphertext of a plaintext msg under pk using the VFE-HW.Enc algorithm as step (9). If an external encryptor generates CT, it is sent to the client. Note that we omit this procedure in Figure 1. In the decryption procedure, the client setups a FE parameterized P in the DNP. Then, the client checks the validity of $\mathsf{sk_P}$ and CT using the VFE-HW.VerifyK and VFE-HW.VerifyCT algorithms respectively as step (10). If $\mathsf{sk_P}$ and CT are valid, the client inputs CT, P and $\mathsf{sk_P}$ into the FE via hardware invocation as step (11). If the DNP is managed remotely by the client, then a remote attestation is employed in this case. Next, the FE transfers $\mathsf{sk_P}$ to the DE via a local attestation as step (12). The validity of $\mathsf{sk_P}$ is confirmed by using the SIG.Verify algorithm as step (13). If $\mathsf{sk_P}$ is valid, the DE transfers $\mathsf{dk_{vpke}}$ to FE via a local attestation as step (14). The FE decrypts CT as step (15) using the aVPKE.Conv and $\mathsf{VPKE.Dec'}$ algorithms. Finally, the client obrtains $\mathsf{P(msg)}$ as step (16).

6 Security Analysis

We provide two proofs to demonstrate that the proposed scheme provides weak verifiability and simulation security.

6.1 Weak Verifiability

In this section, we prove the weak verifiability of VFE-HW. Essencially, we employ the strictly non-trivial public verifiability of VPKE. To do so, we need to guarantee that $\mathsf{dk_{vpke}}$ used in the VPKE.Dec algorithm is generated correctly by the VPKE.KeyGen algorithm. We guarantee this using the correctness of HW. Formally, the following theorem holds.

Theorem 2 VFE-HW *is weak verifiable if* VPKE *is strictly non-trivial public verifiable, and* HW *is correct.*

Proof. According to our trusted setup assumption, $\mathsf{VFE\text{-}HW.Setup^{HW}}$ and $\mathsf{VFE\text{-}HW.DecSetup^{HW,KM(\cdot)}}$ algorithms were honestly run which means that $\mathsf{dk_{vpke}}$ was correctly generated, and sent from the KME to a DE. Moreover, $\mathsf{VFE\text{-}HW.VerifyCT(mpk, CT)} = 1$ and $\mathsf{VFE\text{-}HW.VerifyK(mpk, P, sk_P)} = 1$ hold. Now, we need to guarantee that $\mathsf{dk_{vpke}}$ is correctly sent from the DE to a FE in the $\mathsf{VFE\text{-}HW.Dec^{HW(\cdot)}}$ algorithm. This holds with probability $1 - \mathsf{negl}(\lambda)$ due to the correctness of HW. Next, by using this $\mathsf{dk_{vpke}}$, $\mathsf{VPKE.Ver(pars, pk_{vpke}, CT)} = 1 \Rightarrow \mathsf{VPKE.Dec(pars, pk_{vpke}, dk_{vpke}, CT)} \neq \perp$ holds due to the strictly non-trivial public verifiability of VPKE. Thus, decryption result of CT is determined to be unique since the VPKE.Dec algorithm is deterministic algorithm. Let the decryption result denote msg. Then, the VFE-HW.Dec algorithm outputs $\mathsf{P(msg)}$ from P and msg.

6.2 Simulation Security

Here, we prove the simulation security of the VFE-HW scheme. We replace the PKE scheme of IRON with a VPKE scheme. In this case, we primarily consider whether the SIM-based security is preserved after the replacement. In other words, an adversary $\mathcal{A}$ can check the validity of ciphertexts and it may use for distinguishing REAL and IDEAL. For example, if the challenge ciphertext is changed as a random number (typically employed to provide key privacy/anonymity in the PKE/IBE context), then the public verifiability helps $\mathcal{A}$ to distinguish REAL and IDEAL, and the proof fails. Fortunately, the security proof of IRON does not employ the step, and hence we can replace the PKE scheme with the VPKE scheme.

Theorem 3 $\mathsf{VFE\text{-}HW}$ *is simulation secure if* VPKE *is IND-CCA secure,* PKE *is IND-CCA secure,* SIG *is EUF-CMA secure, and* HW *is a secure hardware scheme.*

Proof. We construct a simulator $\mathcal{S}$. First, $\mathcal{S}$ needs to simulate the Pre-Processing phase as REAL. $\mathcal{S}$ runs $\mathsf{HW.Setup}(1^\lambda)$ and records $(\mathsf{sk_{report}}, \mathsf{sk_{quote}})$. $\mathcal{S}$ measures the designated program $\mathsf{Q_{DE}}$, and stores the program tag $\mathsf{tag}_{\mathsf{Q_{DE}}}$. Finally, $\mathcal{S}$ creates seven empty lists $\mathcal{L}_K, \mathcal{L}_R, \mathcal{L}_D, \mathcal{L}_{KM}, \mathcal{L}_{DE}, \mathcal{L}_{DE2}$, and $\mathcal{L}_{FE}$.

We use sequences of games Game_0, ... , Game_7 to prove that adversary $\mathcal{A}$ cannot computationally distinguish between REAL and IDEAL as follows.

$\boxed{\text{Game}_0}$ $\mathcal{S}$ runs REAL.

$\boxed{\text{Game}_1}$ $\mathcal{S}$ runs as Game_0 with the following exceptions

- $\mathsf{HW.LOAD}(\mathsf{params}, \mathsf{Q_{DE}})$: If $\mathcal{A}$ queries this oracle as input params and $\mathsf{Q_{DE}}$, $\mathcal{S}$ responds $\mathsf{hdl_{DE}}$ by running the $\mathsf{HW.Load}(\mathsf{params}, \mathsf{Q_{DE}})$ algorithm, and storing it in $\mathcal{L}_D$.

- $\mathsf{HW.LOAD}(\mathsf{params}, \mathsf{Q_{FE}}(\mathsf{P}))$: If $\mathcal{A}$ queries this oracle as input params and $\mathsf{Q_{FE}}(\mathsf{P})$, $\mathcal{S}$ responds $\mathsf{hdl_P}$ by running the $\mathsf{HW.Load}(\mathsf{params}, \mathsf{Q_{FE}}(\mathsf{P}))$ algorithm, and storing it in $\mathcal{L}_K$. If $\mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})} \notin \mathcal{L}_K$, then $\mathcal{S}$ stores $(0, \mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})}, \mathsf{hdl_{FE}}(\mathsf{P}))$ in $\mathcal{L}_K$.

- $\mathsf{HW.RUN}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input hdl and in, $\mathcal{S}$ responds out by running the $\mathsf{HW.Run}(\mathsf{hdl}, \mathsf{in})$ algorithm. If $\mathsf{vk_{sign}}$, which is queried by $\mathcal{A}$ as the $\mathsf{HW.Run}(\mathsf{hdl_{DE}}, \mathsf{in} =$ (“init setup”, $\mathsf{vk_{sign}}$)) algorithm, is not the same as that of mpk, $\mathcal{S}$ removes $\mathsf{hdl_{DE}}$ from $\mathcal{L}_D$.

- $\mathsf{VFE\text{-}HW.KeyGen}^{\mathsf{HW}}(\mathsf{msk}, \mathsf{P})$: If $\mathcal{A}$ queries to this oracle as input P, $\mathcal{S}$ responds $\mathsf{sk_P}$ by running the $\mathsf{HW.Run}(\mathsf{hdl}, \mathsf{in})$ algorithm as follows. Parse $\mathsf{msk} = \mathsf{hdl_{KME}}$. $\mathcal{S}$ computes $\mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})}$, calls $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{KME}}$, (“sign”, $\mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})}$)), and outputs $\mathsf{sk_P} := \mathsf{sig}$. If $\mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})}$ already has an entry in $\mathcal{L}_K$, $\mathcal{S}$ creates the first entry 1 (we call “honest-bit” for the first entry in $\mathcal{L}_K$); otherwise, $\mathcal{S}$ adds the tuple $(1, \mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})}, \{\})$ to $\mathcal{L}_K$.

- $\mathsf{VFE\text{-}HW.Enc}(\mathsf{mpk}, \mathsf{msg})$: If $\mathcal{A}$ queries this encryption algorithm as input msg, $\mathcal{S}$ responds CT by running the $\mathsf{VPKE.Enc}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{msg})$ algorithm. If msg is a challenge plaintext msg^*, $\mathcal{S}$ responds $\mathsf{CT}^*$by running the algorithm, and stores it in $\mathcal{L}_R$.

$\boxed{\text{Game}_2}$ $\mathcal{S}$ runs as Game_1 with the following exceptions.

$\mathsf{HW.RUN\&REPORT}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl_{DE}}$ and $\mathsf{in} =$ (“provision”, report, sig), then $\mathcal{S}$ responds $\mathsf{report_{dk}}$ by running the $\mathsf{HW.Run\&Report}_{\mathsf{sk_{report}}}(\mathsf{hdl_{DE}}$, (“provision”, report, sig)) algorithm. If $\mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})}$ in report is not contained as a component of an honest-bit tuple in $\mathcal{L}_K$, $\mathcal{S}$ outputs $\perp$.

Here, we consider a case where the $\mathsf{HW.RUN\&REPORT}(\mathsf{hdl_{DE}}$, (“provision”, $\mathsf{report}, \mathsf{sig}$)) algorithm outputs non $\perp$ even if $\mathsf{tag}_{\mathsf{Q_{FE}}(\mathsf{P})}$ is not contained as an honest-bit tuple in $\mathcal{L}_K$. If $\mathcal{A}$ can make a query while ensuring this case, we can break the existentially unforgeability for SIG with non-negligible probability. The following Lemma is the same as Lemma C.1 of IRON.

Lemma 1 *If the signature scheme* SIG *is EUF-CMA secure, then* Game_2 *is indistinguishable from* Game_1*.*

$\boxed{\text{Game}_{3.0}}$ $\mathcal{S}$ runs as Game_2 with the following exceptions.

1. $\mathsf{HW.RUN\&QUOTE}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{DE}}$ and $\mathsf{in} = $ (“init setup”, $\mathsf{vk}_{\mathsf{sign}}$), $\mathcal{S}$ responds quote by running the $\mathsf{HW.Run\&Quote}_{\mathsf{sk}_{\mathsf{quote}}}(\mathsf{hdl}_{\mathsf{DE}}$, (“init setup”, $\mathsf{vk}_{\mathsf{sign}}$)) algorithm, and stores $\mathsf{out} = (\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}})$ as a component of quote in $\mathcal{L}_{DE2}$.

2. $\mathsf{HW.RUN}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{KME}}$ and $\mathsf{in} = $ (“provision”, $\mathsf{quote}, \mathsf{params}$), $\mathcal{S}$ responds $(\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}, \sigma_{\mathsf{dk}})$ by running the $\mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{KME}}$, (“provision”, quote, params)) algorithm. If $(\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}}) \notin \mathcal{L}_{DE2}$, then $\mathcal{S}$ outputs $\perp$.

Here, we consider a case where the $\mathsf{HW.RUN}(\mathsf{hdl}_{\mathsf{KME}}$, (“provision”, $\mathsf{quote}, \mathsf{params}$)) algorithm outputs non $\perp$ even if $(\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}}) \notin \mathcal{L}_{DE2}$. Here, if $\mathcal{A}$ can make a query while ensuring this case, then we can break the remote attestation unforgeability for HW with non-negligible probability. The following Lemma is the same as Lemma C.4 of IRON.

Lemma 2 *If the secure hardware scheme* HW *is REM-ATT-UNF secure, then* $\text{Game}_{3.0}$ *is indistinguishable from* Game_2.

$\boxed{\text{Game}_{3.1}}$ $\mathcal{S}$ runs as $\text{Game}_{3.0}$ with the following exceptions.

1. $\mathsf{HW.RUN\&REPORT}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{FE}}(\mathsf{P})$ and $\mathsf{in} = $ “init”, then $\mathcal{S}$ responds report by running the $\mathsf{HW.Run\&Report}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}_{\mathsf{FE}}(\mathsf{P})$, “init”) algorithm, and storing $\mathsf{out} = (\mathsf{sid}, \mathsf{pk}_{\mathsf{la}})$ as a component of report in $\mathcal{L}_{FE}$.

2. $\mathsf{HW.RUN}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{DE}}$ and $\mathsf{in} = $ (“provision”, $\mathsf{report}, \mathsf{sig}$), $\mathcal{S}$ responds $\mathsf{report}_{\mathsf{dk}}$ by running the $\mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{DE}}$, (“provision”, $\mathsf{report}, \mathsf{sig}$)) algorithm. If $(\mathsf{sid}, \mathsf{pk}_{\mathsf{la}}) \notin \mathcal{L}_{FE}$, $\mathcal{S}$ outputs $\perp$.

Here, we consider a case where the $\mathsf{HW.RUN\&REPORT}(\mathsf{hdl}_{\mathsf{DE}}$, (“provision”, $\mathsf{report}, \mathsf{sig}$)) algorithm outputs non $\perp$ even if $(\mathsf{sid}, \mathsf{pk}_{\mathsf{la}}) \notin \mathcal{L}_{FE}$. If $\mathcal{A}$ can make a query while ensuring this case, we can break the local attestation unforgeability for HW with non-negligible probability. The following Lemma is the same as Lemma C.5 of IRON.

Lemma 3 *If the secure hardware scheme* HW *is LOC-ATT-UNF secure,* $\text{Game}_{3.1}$ *is indistinguishable from* $\text{Game}_{3.0}$.

$\boxed{\text{Game}_{4.0}}$ $\mathcal{S}$ runs as $\text{Game}_{3.1}$ with the following exceptions.

$\mathsf{HW.RUN}(\mathsf{hdl}, \mathsf{in})$:

1. If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{KME}}$ and $\mathsf{in} = $ (“provision”, $\mathsf{quote}, \mathsf{params}$), $\mathcal{S}$ responds $(\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}})$ by running the $\mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{KME}}$, (“provision”, $\mathsf{quote}, \mathsf{params}$)) algorithm, and storing it in $\mathcal{L}_{KM}$.
2. If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{DE}}$ and $\mathsf{in} = $ (“complete setup”, $\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}$, σ_{dk}), $\mathcal{S}$ runs the $\mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{DE}}$, (“complete setup”, $\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}$)) algorithm. If $(\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}) \notin \mathcal{L}_{KM}$, then $\mathcal{S}$ outputs $\perp$.

Here, we consider a case that the $\mathsf{HW.RUN}(\mathsf{hdl}_{\mathsf{DE}}$, (“complete setup”, $\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}, \sigma_{\mathsf{dk}}$)) algorithm outputs non $\perp$ even if $(\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}) \notin \mathcal{L}_{KM}$. If $\mathcal{A}$ can make a query while ensuring this case, we can break the existentially unforgeability for SIG with non-negligible probability. The following Lemma is the same as Lemma C.2 of IRON.

Lemma 4 *If the signature scheme* SIG *is EUF-CMA secure,* $\text{Game}_{4.0}$ *is indistinguishable from* $\text{Game}_{3.1}$*.*

$\boxed{\text{Game}_{4.1}}$ $\mathcal{S}$ runs as $\text{Game}_{4.0}$ with the following exceptions.

1. $\mathsf{HW.RUN\&REPORT}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{DE}}$ and in = (“provision”, $\mathsf{report}, \mathsf{sig}$), $\mathcal{S}$ responds $\mathsf{report}_{\mathsf{dk}}$ by running the $\mathsf{HW.Run\&Report}_{\mathsf{sk}_{\mathsf{report}}}(\mathsf{hdl}_{\mathsf{DE}}$, (“provision”, report, sig)) algorithm, and storing $\mathsf{out} = (\mathsf{sid}, \mathsf{ct}_{\mathsf{key}})$ as a component of $\mathsf{report}_{\mathsf{dk}}$ in $\mathcal{L}_{DE}$.

2. $\mathsf{HW.RUN}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{FE}}(\mathsf{P})$ and in = (“run”, $\mathsf{params}, \mathsf{mpk}, \mathsf{pk}_{\mathsf{la}}, \mathsf{report}_{\mathsf{dk}}, \mathsf{CT}$), $\mathcal{S}$ responds $\mathsf{P}(\mathsf{msg})$ by running the $\mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{FE}}(\mathsf{P})$, (“run”, $\mathsf{params}, \mathsf{mpk}, \mathsf{pk}_{\mathsf{la}}, \mathsf{report}_{\mathsf{dk}}, \mathsf{CT}$)) algorithm. If $(\mathsf{sid}, \mathsf{ct}_{\mathsf{key}}) \notin \mathcal{L}_{DE}$, $\mathcal{S}$ outputs $\perp$.

Here, we consider a case where the $\mathsf{HW.RUN}(\mathsf{hdl}_{\mathsf{P}}$, (“run”, $\mathsf{params}, \mathsf{mpk}, \mathsf{pk}_{\mathsf{la}}, \mathsf{report}_{\mathsf{dk}}, \mathsf{CT}$)) algorithm outputs non $\perp$ even if $(\mathsf{sid}, \mathsf{ct}_{\mathsf{key}}) \notin \mathcal{L}_{DE}$. If $\mathcal{A}$ can make a query while ensuring this case, we can break the local attestation unforgeability for HW with non-negligible probability. The following Lemma is the same as Lemma C.3 of IRON.

Lemma 5 *If the secure hardware scheme* HW *is LOC-ATT-UNF secure,* $\text{Game}_{4.1}$ *is indistinguishable from* $\text{Game}_{4.0}$*.*

$\boxed{\text{Game}_{5}}$ $\mathcal{S}$ runs as $\text{Game}_{4.1}$ with the following exceptions.

$\mathsf{HW.RUN}(\mathsf{hdl}, \mathsf{in})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{hdl} = \mathsf{hdl}_{\mathsf{FE}}(\mathsf{P})$ and in = (“run”, $\mathsf{params}, \mathsf{mpk}, \mathsf{pk}_{\mathsf{la}}, \mathsf{report}_{\mathsf{dk}}, \mathsf{CT}$), $\mathcal{S}$ evaluates CT as follows.

- If $\mathsf{CT} \notin \mathcal{L}_{\mathsf{R}}$, $\mathcal{S}$ retrieves $\mathsf{dk}_{\mathsf{vpke}}$ from $\mathsf{ct}_{\mathsf{key}}$, and computes $\mathsf{msg} \leftarrow \mathsf{VPKE.Dec}(\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, \mathsf{dk}_{\mathsf{vpke}}, \mathsf{CT})$. Finally, $\mathcal{S}$ evaluates P on msg, and outputs $\mathsf{out} := \mathsf{P}(\mathsf{msg})$
- If $\mathsf{CT} \in \mathcal{L}_{\mathsf{R}}$, $\mathcal{S}$ uses the $\mathsf{U}_{\mathsf{msg}^*}(\mathsf{P})$ oracle, and responds with $\mathsf{P}(\mathsf{msg}^*)$.

$\boxed{\text{Game}_{6}}$ $\mathcal{S}$ runs as Game_{5} with the following exceptions.

$\mathsf{KM}(\mathsf{quote})$: If $\mathcal{A}$ queries this oracle as input $\mathsf{quote} = (\mathsf{md}_{\mathsf{hdl}_{\mathsf{DE}}}, \mathsf{tag}_{\mathsf{Q}_{\mathsf{DE}}}, \mathsf{in}$ = (“run”, $\mathsf{vk}_{\mathsf{sign}}$), $\mathsf{out} = (\mathsf{sid}, \mathsf{pk}_{\mathsf{ra}}), \sigma)$, $\mathcal{S}$ runs the $\mathsf{HW.Run}(\mathsf{hdl}_{\mathsf{KME}}$, (“provision”, $\mathsf{quote}, \mathsf{params}$)) algorithm, which internally runs $\mathsf{ct}_{\mathsf{dk}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{ra}}, 0^{|\mathsf{dk}_{\mathsf{vpke}}|})$, and outputs $(\mathsf{sid}, \mathsf{ct}_{\mathsf{dk}}, \sigma_{\mathsf{dk}})$.

The following Lemma is the same as Lemma C.6 of IRON.

Lemma 6 *If the public key encryption scheme* PKE *is IND-CCA secure,* Game_{6} *is indistinguishable from* Game_{5}*.*

$\boxed{\text{Game}_{7}}$ $\mathcal{S}$ runs as Game_{6} with the following exceptions.

$\mathsf{VFE\text{-}HW.Enc}(\mathsf{mpk}, 0^{|\mathsf{msg}^*|})$: If $\mathcal{A}$ queries this algorithm as input msg, $\mathcal{S}$ responds CT by running $\mathsf{VPKE.Enc}(\mathsf{pars}, \mathsf{pk}_{\mathsf{vpke}}, 0^{|\mathsf{msg}|})$. If msg is a challenge plaintext msg^*, $\mathcal{S}$ responds CT^* by running the algorithm, and storing it in $\mathcal{L}_{R}$.

Here, no step replaces a valid ciphertext with an invalid ciphertext, e.g., a random number; therefore, the public verifiability does not affect the security proof.

Lemma 7 *If the verifiable public key encryption scheme* VPKE *is IND-CCA secure,* Game_7 *is indistinguishable from* Game_6*.*

Proof. Let $\mathcal{A}$ be an adversary who distinguishes between Game_6 and Game_7, and let$\mathcal{C}$ be the challenger of IND-CCA security. We construct an algorithm $\mathcal{B}$ that breaks IND-CCA as follows. First, $\mathcal{C}$ runs $\mathsf{pars} \leftarrow \mathsf{VPKE.PGen}(1^\lambda)$, then $(\mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}) \leftarrow \mathsf{VPKE.KeyGen}(\mathsf{pars})$, and gives pars and $\mathsf{pk_{vpke}}$ to $\mathcal{B}$. $\mathcal{B}$ runs $(\mathsf{sk_{sign}}, \mathsf{vk_{sign}}) \leftarrow \mathsf{SIG.KeyGen}(1^\lambda)$ and $\mathsf{params} \leftarrow \mathsf{HW.Setup}(1^\lambda)$, and gives params and $\mathsf{mpk} = (\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{vk_{sign}})$ to $\mathcal{A}$.

For key generation query P, $\mathcal{B}$ derives $\mathsf{tag}_{\mathsf{Q_{FE}(P)}}$ from P, and calls $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{KME}}, (\text{“sign”}, \mathsf{tag}_{\mathsf{Q_{FE}(P)}}))$. Then, $\mathcal{B}$ sends $\mathsf{sk_P} := \mathsf{sig}$ to $\mathcal{A}$, and stores $\mathsf{tag}_{\mathsf{Q_{FE}(P)}}$ in $\mathcal{L}_K$.

For run query $(\mathsf{hdl_{FE}}(\mathsf{P}), (\text{“run”}, \mathsf{params}, \mathsf{mpk}, \mathsf{pk_{la}}, \mathsf{report_{dk}}, \mathsf{CT}))$ where $\mathsf{report_{dk}}$ is valid and $\mathsf{hdl_{FE}}(\mathsf{P}) \in \mathcal{L}_\mathsf{K}$ with honest-bit, $\mathcal{B}$ forwards CT to $\mathcal{C}$ as a decryption query. $\mathcal{C}$ returns msg by running the $\mathsf{VPKE.Dec}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}, \mathsf{CT})$ algorithm to $\mathcal{B}$. If $\mathsf{msg} = \perp$, $\mathcal{B}$ outputs $\perp$; otherwise, $\mathcal{B}$ runs P on msg, and sends $\mathsf{P}(\mathsf{msg})$ to $\mathcal{A}$.

In the challenge phase, $\mathcal{A}$ sends $(\mathsf{msg}^*, 0^{|\mathsf{msg}^*|})$ to $\mathcal{B}$. $\mathcal{B}$ sets $\mathsf{msg}^* = \mathsf{M}_0^*$ and $0^{|\mathsf{msg}^*|} = \mathsf{M}_1^*$, and sends $(\mathsf{M}_0^*, \mathsf{M}_1^*)$ to $\mathcal{C}$. $\mathcal{C}$ computes challenge ciphertext $\mathsf{CT}^* = \mathsf{VPKE.Enc}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{M}_\mu^*)$ where $\mu \in \{0,1\}$, and sends CT^* to $\mathcal{B}$. $\mathcal{B}$ sends CT^* to $\mathcal{A}$, and stores CT^* in $\mathcal{L}_R$.

For key generation query P, $\mathcal{B}$ derives $\mathsf{tag}_{\mathsf{Q_{FE}(P)}}$ from P, and calls $\mathsf{sig} \leftarrow \mathsf{HW.Run}(\mathsf{hdl_{KME}}, (\text{“sign”}, \mathsf{tag}_{\mathsf{Q_{FE}(P)}}))$. $\mathcal{B}$ sends $\mathsf{sk_P} := \mathsf{sig}$ to $\mathcal{A}$, and stores $\mathsf{tag}_{\mathsf{Q_{FE}(P)}}$ in $\mathcal{L}_K$.

For run query $(\mathsf{hdl_{FE}}(\mathsf{P}), (\text{“run”}, \mathsf{params}, \mathsf{mpk}, \mathsf{pk_{la}}, \mathsf{report_{dk}}, \mathsf{CT}))$ where $\mathsf{report_{dk}}$ is valid and $\mathsf{hdl_P} \in \mathcal{L}_\mathsf{K}$ with honest-bit:

- $\mathsf{CT} \in \mathcal{L}_\mathsf{R}$: $\mathcal{B}$ uses the universal oracle $\mathsf{U}_{\mathsf{msg}^*}(\mathsf{P})$, and sends $\mathsf{P}(\mathsf{msg}^*)$ to $\mathcal{A}$.
- $\mathsf{CT} \notin \mathcal{L}_\mathsf{R}$: $\mathcal{B}$ forwards CT to $\mathcal{C}$ as a decryption query. $\mathcal{C}$ returns msg by running the $\mathsf{VPKE.Dec}(\mathsf{pars}, \mathsf{pk_{vpke}}, \mathsf{dk_{vpke}}, \mathsf{CT})$ algorithm to $\mathcal{B}$. If $\mathsf{msg} = \perp$, $\mathcal{B}$ outputs $\perp$; otherwise, $\mathcal{B}$ runs P on msg, and sends $\mathsf{P}(\mathsf{msg})$ to $\mathcal{A}$.

Finally, $\mathcal{A}$ outputs $\mu' \in \{0,1\}$. $\mathcal{B}$ outputs μ', and breaks IND-CCA security.

7 Implementation

In this section, we give an implementation result when we employ a cryptographic hash function H as a function P, i.e., the decryption algorithm outputs $\mathsf{H}(\mathsf{msg})$. As mentioned before, theoretically the function is not realized in the IND-based VFE scheme [10] due to the collision-resistance of H, and practically the function seems attractive when we compute a hashed value for a sensitive data such as a password. This system can be achieved by IRON, however no verifiability is guaranteed. On the other hand, in our scheme the server can verify the ciphertext, and can delegate the verification to another server as an option.

We measured the average times and standard deviations of the $\mathsf{VFE\text{-}HW.Enc}$, $\mathsf{VFE\text{-}HW.VerifyCT}$, $\mathsf{VFE\text{-}HW.VerifyK}$ and $\mathsf{VFE\text{-}HW.Dec}$ algorithms because we estimate the runtime of the algorithms related to msg for the proposed scheme. Here, except for the $\mathsf{VFE\text{-}HW.Dec}$ algorithm, all algorithms were run outside enclaves. In the $\mathsf{VFE\text{-}HW.Dec}$ algorithm, the FE runs the $\mathsf{VPKE.Conv}$ and $\mathsf{VPKE.Dec'}$ algorithms, and evaluates H on msg. We employ the VPKE scheme [33], ECDSA as SIG, and SHA-256 as H.

The $\mathsf{VPKE.Ver}$ algorithm checks whether (part of) the ciphertext is a DDH tuple, we employed symmetric pairings even though asymmetric pairings are desirable for efficient implementation [25]. We used the PBC library [1], which supports the symmetric pairings. We generated parameters for a Type-A curve with 128-bit security, defined over the field $\mathbb{F}_p$ with a 256-bit prime p, where the

Table 2: Implementation results of VFE-HW scheme

Running Time (sec)	Average	Standard Deviation
VFE-HW.Enc	0.12436	0.00250
VFE-HW.VerifyCT	0.12828	0.00259
VFE-HW.VerifyK	0.00034	0.00005
VFE-HW.Dec	0.06499	0.00163

Table 3: Implementation results of VFE-HW scheme (Invalid ciphertext/secret key)

Running Time (sec)	Average	Standard Deviation
VFE-HW.VerifyCT (DDH)	0.11828	0.00228
VFE-HW.VerifyCT (OTS)	0.12329	0.00252
VFE-HW.VerifyK (Signature)	0.00034	0.00006

order is a 1536-bit prime, using a function called `pbc_param_init_a_gen`. The parameters is given in Appendix B. For running the PBC library in enclaves, we employed the PBC for SGX given by Contiu et al. [20]. In our implementation, we set the input-output of enclaves is as an array of unsigned char values regarding a valuable of PBC. We transformed the binary data into an element of elliptic curves using the `element_from_bytes` function supported by PBC within enclaves.

Our implementation environment includes the CPU: Intel(R) Core(TM) i3-7100U (2.40GHz), and the libraries openssl 1.0.2g, Intel SGX 1.5 Linux Driver, Intel SGX SDK, Intel SGX PSW, GMP, PBC, and PBC for SGX [20].

We give our implementation result in Table 2. Compared to the running time of the VFE-HW.Dec algorithm, which was run inside the enclave, those of the VFE-HW.Enc and VFE-HW.VerifyCT algorithms were relatively slow. The reason seems to employ symmetric bilinear groups in our implementation, i.e., the size of the group $\mathbb{G}$ is much larger than that of the case of asymmetric bilinear groups. Thus, proposing a VPKE scheme secure in asymmetric bilinear groups (or without pairings) and re-implementing our VFE-HW scheme seems an interesting future work. Since we focus on verifiability of ciphertexts and secret keys, we also evaluate when VFE-HW.VerifyCT and VFE-HW.VerifyK algorithms output 0 in Table 3. In our implementation, the VFE-HW.VerifyCT algorithm outputs 0 either the DDH test or a verification of One-Time Signature (OTS) [39] fails. The VFE-HW.VerifyK algorithm outputs 0 when a verification of signature fails. Even if the verification process fails when invalid ciphertexts or secret keys are used, the running times are similar to those of valid ciphertexts or secret keys.

8 Conclusion

In this paper, we proposed a SIM-based VFE that supports any functionality. To support any functionality, we employed a hardware-based construction. In addition, we gave a SIM-based VFE construction that employs VPKE, PKE, SIG, and HW. Finally, we give our implementation of proposed VFE-HW scheme for H. Recently, Bhatotia et al. [12] considered a composable security when Trusted Execution Environments (TEEs) including Intel SGX are employed. Considering such a composability in the VFE-HW context is left as a future work. Although we have claimed that the trusted assumption is reasonable in the HW setting, we leave how to remove this assumption without losing the SIM-based security as a future work. In addition, we leave how to construct SIM-based secure VFE without using secure hardware as a future work.

Acknowledgement

This work was supported by the JSPS KAKENHI Grant Numbers JP20K11811, JP20J22324, and JP21K11897. We thank Dr. Rafael Pires for helpful discussion.

References

[1] The pbc (pairing-based cryptography) library. available at http://crypto.stanford.edu/pbc/.

[2] M. Abdalla, F. Bourse, A. D. Caro, and D. Pointcheval. Simple functional encryption schemes for inner products. In *PKC*, pages 733–751, 2015.

[3] M. Abdalla, F. Bourse, H. Marival, D. Pointcheval, A. Soleimanian, and H. Waldner. Multi-client inner-product functional encryption in the random-oracle model. In *SCN*, pages 525–545, 2020.

[4] M. Abdalla, D. Catalano, R. Gay, and B. Ursu. Inner-product functional encryption with fine-grained access control. In *ASIACRYPT*, pages 467–497, 2020.

[5] M. Abdalla, J. Gong, and H. Wee. Functional encryption for attribute-weighted sums from k-lin. In *CRYPTO*, pages 685–716, 2020.

[6] S. Agrawal, S. Gorbunov, V. Vaikuntanathan, and H. Wee. Functional encryption: New perspectives and lower bounds. In *CRYPTO*, pages 500–518, 2013.

[7] S. Agrawal, V. Koppula, and B. Waters. Impossibility of simulation secure functional encryption even with random oracles. In *TCC*, pages 659–688, 2018.

[8] S. Agrawal, B. Libert, M. Maitra, and R. Titiu. Adaptive simulation security for inner product functional encryption. In *PKC*, pages 34–64, 2020.

[9] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.

[10] S. Badrinarayanan, V. Goyal, A. Jain, and A. Sahai. Verifiable functional encryption. In *ASIACRYPT*, pages 557–587, 2016.

[11] B. Barak and R. Pass. On the possibility of one-message weak zero-knowledge. In *TCC*, pages 121–132, 2004.

[12] P. Bhatotia, M. Kohlweiss, L. Martinico, and Y. Tselekounis. Steel: Composable hardware-based stateful and randomised functional encryption. In *PKC*, pages 709–736, 2021.

[13] M. Blum, P. Feldman, and S. Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *STOC*, pages 103–112, 1988.

[14] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In *TCC*, pages 253–273, 2011.

[15] E. Boyle, K. Chung, and R. Pass. On extractability obfuscation. In *TCC*, pages 52–73, 2014.

[16] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX*, pages 991–1008, 2018.

[17] J. V. Bulck, D. Moghimi, M. Schwarz, M. Lipp, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *IEEE S&P*, pages 54–72, 2020.

[18] A. D. Caro, V. Iovino, A. Jain, A. O'Neill, O. Paneth, and G. Persiano. On the achievability of simulation-based security for functional encryption. In *CRYPTO*, pages 519–535, 2013.

[19] J. Chotard, E. Dufour Sans, R. Gay, D. H. Phan, and D. Pointcheval. Decentralized multi-client functional encryption for inner product. In *ASIACRYPT*, pages 703–732, 2018.

[20] S. Contiu, R. Pires, S. Vaucher, M. Pasin, P. Felber, and L. Réveillère. IBBE-SGX: cryptographic group access control using trusted execution environments. In *DSN*, pages 207–218, 2018.

[21] I. Damgård, H. Haagh, R. Mercer, A. Nitulescu, C. Orlandi, and S. Yakoubov. Stronger security and constructions of multi-designated verifier signatures. In *TCC*, pages 229–260, 2020.

[22] P. Datta, T. Okamoto, and J. Tomida. Full-hiding (unbounded) multi-input inner product functional encryption from the k-linear assumption. In *PKC*, pages 245–277, 2018.

[23] E. Dufour Sans and D. Pointcheval. Unbounded inner-product functional encryption with succinct keys. In *ACNS*, pages 426–441, 2019.

[24] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. IRON: functional encryption using intel SGX. In *ACM CCS*, pages 765–782, 2017.

[25] S. D. Galbraith, K. G. Paterson, and N. P. Smart. Pairings for cryptographers. In *Discrete Applied Mathematics*, pages 3113–3121, 2008.

[26] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM J. Comput.*, 45(3):882–929, 2016.

[27] S. Garg, C. Gentry, S. Halevi, and M. Zhandry. Fully secure attribute based encryption from multilinear maps. *IACR Cryptology ePrint Archive*, 2014:622, 2014.

[28] O. Goldreich and Y. Oren. Definitions and properties of zero-knowledge proof systems. *J. Cryptology*, 7(1):1–32, 1994.

[29] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM CCS*, pages 89–98, 2006.

[30] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. D. Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP*, 2013.

[31] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel software guard extensions: Epid provisioning and attestation services. 2016.

[32] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP 2013*, 2013.

[33] J. M. G. Nieto, M. Manulis, B. Poettering, J. Rangasamy, and D. Stebila. Publicly verifiable ciphertexts. In *SCN*, pages 393–410, 2012.

[34] A. O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[35] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *ACM CCS*, pages 753–768, 2019.

[36] N. Soroush, V. Iovino, A. Rial, P. B. Rønne, and P. Y. A. Ryan. Verifiable inner product encryption scheme. In *PKC*, pages 65–94, 2020.

[37] J. Tomida and K. Takashima. Unbounded inner product functional encryption from bilinear maps. In *ASIACRYPT*, pages 609–639, 2018.

[38] B. Waters. A punctured programming approach to adaptively secure functional encryption. In *CRYPTO*, pages 678–697, 2015.

[39] H. Wee. Public key encryption against related key attacks. In *PKC*, pages 262–279, 2012.

[40] H. Wee. Attribute-hiding predicate encryption in bilinear groups, revisited. In *TCC*, pages 206–233, 2017.

A The Nieto et al. VPKE scheme

In this appendix, we introduce the Nieto et al. VPKE scheme [33, FIGURE4] as follows. For the underlying One-Time Signature (OTS) scheme, we employ the discrete-log-based Wee OTS scheme [39], and for the DDH test, we employ symmetric pairings whether $\mathrm{e}(g,\, \pi)$ is the same as $\mathrm{e}(c_1,\, u^t v)$ or not.

$\mathsf{VPKE.PGen}(1^\lambda)$: Choose $(p,\, e,\, g,\, \mathbb{G},\, \mathbb{G}_T)$ where $\mathbb{G}$ and $\mathbb{G}_T$ are groups of λ-bit prime order p, $g \in \mathbb{G}$ is a generator, and $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ is a bilinear map. Let $H : \mathbb{G} \rightarrow \{0,1\}^{\mathsf{poly}(\lambda)}$, $H_{OTS} : \{0,1\}^* \rightarrow \{0,1\}^{\mathsf{poly}(\lambda)}$, and $\mathsf{TCR} : \mathbb{G} \times \{0,1\} \rightarrow \mathbb{Z}_p$ be collision or target collision resistant hash functions where $\mathsf{poly}(\lambda)$ is a polynomial in λ. Output $\mathsf{pars} = (p,\, e,\, g,\, \mathbb{G},\, \mathbb{G}_T,\, H,\, H_{OTS},\, TCR)$.

$\mathsf{VPKE.KeyGen}(\mathsf{pars})$: Parse $\mathsf{pars} = (p,\, e,\, g,\, \mathbb{G},\, \mathbb{G}_T,\, H,\, H_{OTS},\, TCR)$. Choose $x_1 \xleftarrow{\$} \mathbb{Z}_p^*$ and $v \xleftarrow{\$} \mathbb{G}$ and compute $u = g^{x_1}$. Output $\mathsf{pk} = (u, v)$ and $\mathsf{dk} = x_1$.

$\mathsf{VPKE.Enc}(\mathsf{pars},\, \mathsf{pk},\, \mathsf{msg})$: Parse $\mathsf{pars} = (p,\, e,\, g,\, \mathbb{G},\, \mathbb{G}_T,\, H,\, H_{OTS},\, TCR)$ and $\mathsf{pk} = (u, v)$. Choose $s_0, s_1, x_2, r, n \xleftarrow{\$} \mathbb{Z}_p^*$ and compute $u_0 = g^{s_0}$, $u_1 = g^{s_1}$, $c' = g^{x_2}$, $c_1 = g^r$, $t \leftarrow TCR(c_1, (u_0, u_1, c'))$, $K \leftarrow H(u^r)$ and $\pi \leftarrow (u^t v)^r$. Set $c_2 \leftarrow \mathsf{msg} \oplus K$ and $c = (c_1, c_2, \pi)$. Compute $w \leftarrow x_2 + n s_0 + s_1(H_{OTS}(c) + n)$. Output $\mathsf{CT} \leftarrow (c, (n, w), (u_0, u_1, c'))$

$\mathsf{VPKE.Ver}(\mathsf{pars},\, \mathsf{pk},\, \mathsf{CT})$: Parse $\mathsf{pars} = (p,\, e,\, g,\, \mathbb{G},\, \mathbb{G}_T,\, H,\, H_{OTS},\, TCR)$, $\mathsf{pk} = (u, v)$, $\mathsf{CT} = (c, (n, w), (u_0, u_1, c'))$ and $c = (c_1, c_2, \pi)$. Compute $t \leftarrow TCR(c_1, (u_0, u_1, c'))$ and $\pi \leftarrow (u^t v)^r$. If $e(g, \pi) \neq e(c_1, u^t v)$ or $g^w \neq c' u_0^n \cdot u_1^{H_{OTS}(c)+n}$, then output 0. Otherwise, output 1.

Table 4: Type A curve with 128-bit security

p	1378291821378419146609392031665627784810724728687992128837360333737763894232758566008499657275579051453797871470115739188384006962567915209697909546472340261341498362791799700699129417020771858468922287416451470375461378349580169934490323687711177168008542310452451285148291313010481717176147391967459404122093602825182059882433251275028588598236180436863368649562718504259977732196012564200822711091269434138471326934527747330048566104052231617611044807535038087
Order	57896044618658097711785492504343953926634992332820282019728792006155588075521
h	23806320975064304888602247447209421656076606270975876064915016694904675238424582942338536744226766065496345901882655664265613708904028566679058218200259833380730762018922498660609790082315613645318317104917054336577361982953438656528379180616414559966902366812187572015942597138104302919587523676824718275034722242569228103402257034633722433381878356381955440717720404013239472452603528
exp1	41
exp2	255
sign0	1
sign1	1

VPKE.Conv: Parse pars $= (p, e, g, \mathbb{G}, \mathbb{G}_T, H, H_{OTS}, TCR)$, pk $= (u, v)$, CT $= (c, (n, w), (u_0, u_1, c'))$ and $c = (c_1, c_2, \pi)$. Output CT$' = (c_1, c_2)$.

VPKE.Dec$'$(pars, pk, dk, CT'): Parse pars $= (p, e, g, \mathbb{G}, \mathbb{G}_T, H, H_{OTS}, TCR)$, pk $= (u, v)$, dk $= x_1$ and CT$' = (c_1, c_2)$. Compute $K \leftarrow H(c_1^{x_1})$ and set msg $\leftarrow c_2 \oplus K$. Output msg.

B Type A Curve with 128-bit Security

Here, we indicate the parameters as shown in Table 4. h is defined as $h := (p + 1)/\mathsf{Order}$ and is a multiple of 12, and sign0, sign1, exp1, and exp2 are defined as $\mathsf{Order} = 2^{\mathsf{exp2}} + \mathsf{sign1} \cdot 2^{\mathsf{exp1}} + \mathsf{sign0} \cdot 1$.