HEKWS: Privacy-Preserving Convolutional Neural Network-based Keyword Spotting with a Ciphertext Packing Technique

Daniel L. Elworth and Sunwoong Kim
Division of Engineering and Mathematics, University of Washington, Bothell, WA 98011, USA
Email: {delworth, sunwoong}@uw.edu

***Abstract*—Keyword spotting (KWS) is a key technology in smart devices. However, privacy issues in these devices have been constantly raised. To solve this problem, this paper applies homomorphic encryption (HE) to a previous small-footprint convolutional neural network (CNN)-based KWS algorithm. This allows for a trustless system in which a command word can be securely identified by a remote cloud server without exposing client data. To alleviate the burden on an edge device of a client, a novel packing technique is proposed that reduces the number of ciphertexts for an input keyword to one. Our HE-based KWS shows a prediction accuracy of 72% for Google's Speech Commands Dataset with 12 labels. This is almost identical to the accuracy of the non-HE-based implementation that has the same CNN layers and approximates a rectified linear unit in the same manner. On a workstation, it takes 19 seconds to process one keyword on average, which can be improved in the future through parallelization, HE parameter optimization, and/or the use of custom hardware accelerators.**

***Index Terms*—keyword spotting, homomorphic encryption, convolutional neural networks, cyber-physical systems**

I. Introduction

As devices in cyber-physical systems (CPS) and/or internet-of-things (IoT) continue to become more ubiquitous, an area of concern is data privacy [1]. With related technologies being used in an increasing number of applications, such as appliances, vehicles, security systems, and health monitors, it is difficult to keep track of information flow and ensure security. The need for improvement in data privacy is apparent, as there has been a massive increase in cyberattacks targeting CPS/IoT devices in the past year. Kaspersky labs reported 1.51 billion breaches of IoT devices in the first half of 2021, over twice the reported number from the same period in 2020 [2].

Keyword spotting (KWS) is a popular CPS/IoT application [3]–[8]. The typical goal of a KWS algorithm is the classification of a spoken word from a predetermined list to detect commands or wake words. One application is the development of voice user interfaces (VUIs) used to improve accessibility options. For instance, VUIs are used to control hearing aids, allowing hands-free adjustments [9]. Another application is a hands-free medical alert system that can be voice-activated to put a user in contact with medical aid in an emergency [10]. By only requiring identification of a small set of words, it can avoid resource-intensive methods used for automatic speech recognition, and thus be used in small-footprint applications. KWS algorithms continuously receive speech data of daily life, including private data such as passwords. If attackers exploit these, serious privacy issues arise. Therefore, *privacy-preserving* KWS algorithms are essential.

Using homomorphic encryption (HE) [11]–[16], a cryptographic technique, allows for a trustless system that eliminates the risk of data exposure. Unlike other techniques that require decryption first to perform operations on a ciphertext, such as the advanced encryption standard (AES) algorithm [17], HE performs operations on ciphertexts directly. This makes it possible to run real-world applications without exposing user data to a third party, a fact relevant to the increasing use of cloud servers [18], [19].

However, HE has a critical problem in that the processing speed is slow [20], [21]. Since the size of each ciphertext containing thousands to tens of thousands of slots is usually several tens or hundreds of MB, the amount of data transmission between server and client is large. In addition, the computational complexity of a client that generates, stores, and transmits ciphertexts is high. Usually, a client is implemented on a resource-hungry embedded system platform with limited processing speed and power consumption, so the large number of ciphertexts is a big burden. To solve this problem, techniques for *packing* multiple plaintext messages into a few ciphertexts and performing slot-wise operations have been widely used [12], [18], [22]–[25]. Such a method greatly reduces the number of ciphertexts and shows good amortized running time. However, it is impossible to perform operations between slots in the same ciphertext without decryption, so how to pack plaintext messages is critical in overall performance [18].

This paper applies a HE scheme to a convolutional neural network (CNN)-based KWS algorithm. We propose a novel packing technique that reduces the number of ciphertexts of speech data and improves slot utilization, which is different from previous HE-based CNN applications [22], [26]. The proposed packing technique moves the client's workload to a server, and therefore it is suitable for a CPS/IoT platform where computation is centered on a resource-rich cloud server.

II. Background

A. CNN-based KWS Algorithms

Several machine learning approaches have been used to implement KWS, including hidden Markov models and vari-

TABLE I
THE HONK COMPACT MODEL [5]

	Conv	ReLU	FC1	FC2	Softmax
m	32	-	-	-	-
r	8	-	-	-	-
n	186	-	128	128	12

ous neural networks [3], [5], [6], [8], with CNNs being one of the most popular. A typical CNN for KWS has one or more convolutional (Conv) layers, followed by several fully connected (FC) layers. At least one activation function, such as a rectified linear unit (ReLU), is used after selected Conv or FC layers. A final FC layer has a node for each target label, and performing the softmax function on the output gives the result of the classification.

A CNN for KWS first requires some preprocessing of an input audio sample. This is done by performing a windowed fast Fourier transform to produce a spectrogram of the signal. The spectrogram can be transformed using the mel scale and have the logarithm of the results found. This becomes the input to a Conv layer of the CNN and has the shape $\mathbf{S} \in \mathbb{R}^{t \times f}$, where f is frequency and t is time. $\mathbf{S}$ is then convolved with weights $\mathbf{W} \in \mathbb{R}^{m \times r \times n}$, where m and r are the frequency and time span of the convolution, and n is the number of feature maps. The process of convolution extracts features from the input. The FC layers consist of nodes. Each node applies separate weights to each input, sums the values and applies a bias. With M inputs, the output of the i-th node is $y_i = b_i + \sum_{n=1}^{M} x_n w_n$, where x, w, and b are the inputs, weights, and biases, respectively.

Some studies use a compact *small-footprint* CNN model rather than a full model for KWS algorithms to minimize parameters and/or multiplications [5], [6]. Table I shows one such compact model presented by Tang and Lin in their Honk reimplementation [5] of a TensorFlow model [27]. Again, m, r, and n in this table represent the width and height of the convolution filter and the number of feature maps or nodes, respectively. This model forms the basis of our HE-based KWS (HEKWS) and is referred to as Honk in the rest of this paper.

B. Operations in HE Schemes

One of the HE schemes commonly implemented in popular open-source HE libraries, such as Microsoft SEAL [28] and PALISADE [29], is the Cheon, Kim, Kim, and Song (CKKS) scheme [11]. This scheme supports homomorphic operations for ciphertexts of fixed-point real-number plaintext messages. The basic operations are as follows (other HE schemes have similar operations except for rescaling (`Rescale`)):

- $\texttt{HomAdd}(\texttt{ct}_0, \texttt{ct}_1)$: adds two ciphertexts. A homomorphic subtraction is also supported.
- $\texttt{HomAddPlain}(\texttt{ct}, \texttt{m})$: adds a plaintext to a ciphertext.
- $\texttt{HomMult}_{\texttt{rk}}(\texttt{ct}_0, \texttt{ct}_1)$: multiplies two ciphertexts. After each multiplication, relinearization (`Relin`) is performed using the relinearization keys `rk` to restore the transformed key to the original one. In addition, `Rescale` is performed to round off plaintext messages in a ciphertext and maintain a scale.
- $\texttt{HomMultPlain}(\texttt{ct}, m)$: multiplies a ciphertext with a plaintext. Unlike `HomMult`, this operation does not require `Relin`.
- $\texttt{HomRot}_{\texttt{gk}}(\texttt{ct})$: rotates a cyclic vector (slots in a ciphertext) by the amount defined by the Galois keys `gk`.

A plaintext message is hidden by noise in a ciphertext, and the noise level increases whenever a homomorphic operation is performed. If it exceeds a certain level, correct results cannot be obtained after decryption. In particular, the noise level significantly increases when performing `HomMult`, and the number of consecutive `HomMult` is defined as the (multiplicative circuit) *depth*. The size of the maximum depth depends on HE parameters.

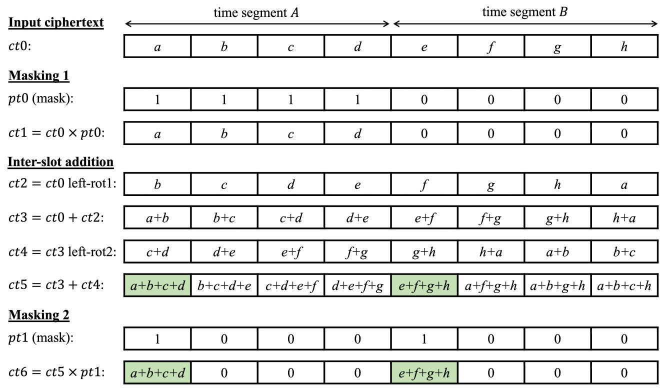
Many HE schemes pack and encode multiple data into a plaintext polynomial, and this polynomial is encrypted as a ciphertext polynomial. Specifically, in the CKKS scheme, a vector of multiple real numbers is encoded. In particular, $N/2$ real numbers are maximally included in $N/2$ slots of a ciphertext, respectively, when the polynomial degree is N. These slots are processed in a single instruction/multiple data (SIMD) manner. For more details about the HE and CKKS scheme, refer to [12].

III. PACKING TECHNIQUES FOR A KEYWORD

In this section, our proposed packing technique in HEKWS is presented. This technique addresses two inherent issues in the HE domain: 1) operations cannot be performed between slots in the same ciphertext, and 2) storing and transmitting a large number of ciphertexts causes issues in performance on the client-side. In order to address these issues, our packing technique creates a single densely-packed ciphertext for a keyword on the client-side and expands the transmitted ciphertext into multiple sparse ciphertexts on the server-side to effectively apply different weights and biases in Conv and FC layers.

A. Client-side Packing

One of the challenges in adapting a CNN for use in a practical HE application is developing an appropriate packing technique to minimize expensive operations and the size of ciphertexts. HEKWS achieves this by maximizing ciphertext slot utilization. For our KWS, the input spectrogram has a shape of (40, 32) in the frequency and time dimensions, respectively, for a total of 1,280 input values. A naïve packing may encode each input value in a separate ciphertext. This has the advantage of requiring no additional homomorphic operations related to packing (e.g., `HomRot`) but wastes resources as only a single slot is utilized per ciphertext. A better approach is to store each time segment (i.e., all the frequency values for a single time point of an input spectrogram) into a separate ciphertext. This approach reduces the required input ciphertexts from 1,280 using the naïve approach to 32 as each ciphertext contains 40 frequency values. However, this approach still requires sending a significant number of ciphertexts from the client. Thus,



	time segment A				time segment B			
Input ciphertext								
$ct0$:	a	b	c	d	e	f	g	h
Masking 1								
$pt0$ (mask):	1	1	1	1	0	0	0	0
$ct1 = ct0 \times pt0$:	a	b	c	d	0	0	0	0
Inter-slot addition								
$ct2 = ct0$ left-rot1:	b	c	d	e	f	g	h	a
$ct3 = ct0 + ct2$:	$a+b$	$b+c$	$c+d$	$d+e$	$e+f$	$f+g$	$g+h$	$h+a$
$ct4 = ct3$ left-rot2:	$c+d$	$d+e$	$e+f$	$f+g$	$g+h$	$h+a$	$a+b$	$b+c$
$ct5 = ct3 + ct4$:	$a+b+c+d$	$b+c+d+e$	$c+d+e+f$	$d+e+f+g$	$e+f+g+h$	$a+f+g+h$	$a+b+g+h$	$a+b+c+h$
Masking 2								
$pt1$ (mask):	1	0	0	0	1	0	0	0
$ct6 = ct5 \times pt1$:	$a+b+c+d$	0	0	0	$e+f+g+h$	0	0	0

Fig. 1. An example of masking and inter-slot addition.

HEKWS maps the spectrogram matrix (frequency $\times$ time) into one dimensional array of size 1,280 and then encrypts it into a densely-packed *single* ciphertext. This calls for an efficient unpacking on the server side to perform homomorhpic operations.

B. Server-side Expansion

The server-side techniques involve three concepts: *masking*, *inter-slot addition*, and *expansion*. Masking is the multiplication of a ciphertext with a plaintext filled with values of either 1 or 0. Masking allows for the separation of time segments onto separate ciphertexts. Fig. 1 shows an example using a simplified input with a dimension of (4, 2). The input ciphertext contains 8 plaintext values (a, b, ..., g, h), and every 4 values belong to each time segment (A or B). To extract the values in A, the input ciphertext is multiplied by the mask plaintext containing (1, 1, 1, 1, 0, 0, 0, 0).

The next concept is inter-slot addition, where the values of a time segment (e.g., a to d) get added together so the weights in CNN can be applied to the ciphertext. As operations between slots in the same ciphertext are not allowed, a copy of a ciphertext is made, rotated, and then added to the original ciphertext. This can be done efficiently by using sequential rotations that increase by a multiple of the previous rotation. Fig. 1 shows an example of inter-slot addition (see the Inter-slot addition and Masking 2 parts). To add up messages from each time segment, the inter-slot addition involves two left-rotations and two additions, and the final results are stored in the zeroth and fourth slots of the ciphertext from the left. Masking is used at the end of the operation to remove artifacts such that only the colored values remain.

Finally, expansion is the process of storing multiple copies of the same time segment on a single ciphertext. The duplicated values allow for different weight positions or values to be applied on the same time segment when processing Conv and FC layers. In our HEKWS design, after receiving the transmitted ciphertext, the resource-rich server uses expansions to create 32 ciphertexts, each containing multiple copies of one time segment. Fig. 2 shows an example of expansion. Expansion is similar to inter-slot addition but includes masking

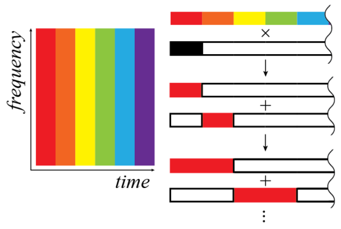
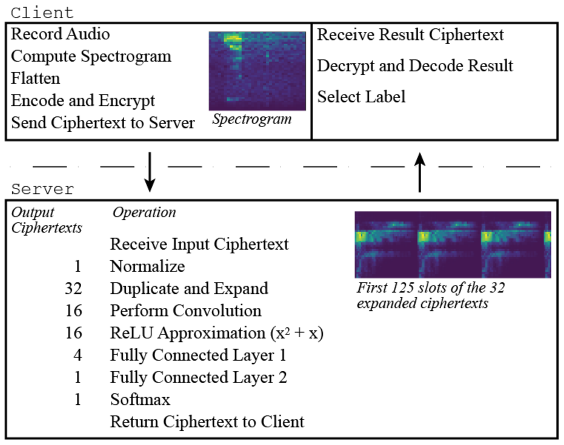


Fig. 2. An example of expansion. The flattened input (a single ciphertext) is masked to extract frequency values for a single window, which are then duplicated to fill slots in the ciphertext by using rotations and additions. This is repeated for each time step (color).



Client
Record Audio
Compute Spectrogram
Flatten
Encode and Encrypt
Send Ciphertext to Server
Spectrogram
Receive Result Ciphertext
Decrypt and Decode Result
Select Label
Server
Output Ciphertexts
Operation
Receive Input Ciphertext
1 Normalize
32 Duplicate and Expand
16 Perform Convolution
16 ReLU Approximation (x² + x)
4 Fully Connected Layer 1
1 Fully Connected Layer 2
1 Softmax
Return Ciphertext to Client
First 125 slots of the 32 expanded ciphertexts

Fig. 3. Process outline for the classification of a command word.

that leaves only the data to be copied and removes the rest. For example, with 4,096 slots and 40 frequency values, there is room for 102 copies on a ciphertext. In order to minimize operations used when processing layers of the CNN, plaintexts are packed to store as many weights and biases as possible to match the expanded ciphertexts.

IV. Network Structure

Our proposed HEKWS is outlined in Fig. 3. Once the server receives the input ciphertext, normalization, $Z = (X - \mu)/\sigma$, is performed on the input ciphertext X, where μ and σ are the mean and standard deviation, respectively. The μ and σ values are constants determined from the training. The normalization stage keeps data centered and mostly contained within a range where the ReLU approximation, discussed later, is valid. After this stage, the ciphertext is expanded into the 32 ciphertexts, as described previously.

After the input ciphertext is expanded into 32 ciphertexts, they are multiplied by the corresponding weights and added together, as shown in Fig. 4 (left). To complete the convolutions, inter-slot addition must be done since each convolution has a frequency span of eight slots. This is done by copying a ciphertext, rotating the copy, adding it to the original, and repeating the process with a rotation twice as large as the previous one. To add the eight adjacent slots, rotations of 1, 2, and 4 are used. In total, 16 sets of weight plaintexts

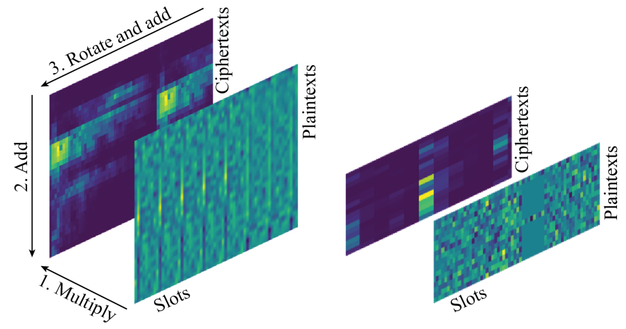


Fig. 4. Application of weight plaintexts (first 60 slots shown) in the Conv layer (left) and the FC1 layer (right).

are applied, giving an output size of 16 ciphertexts from this layer. Bias plaintexts are then added to the ciphertexts, and the expansion process presented in the previous section duplicates values in the ciphertexts for the next layer.

Since HE only supports linear operations, ReLU cannot be directly used in the HE domain. Instead, HEKWS uses $x^2 + x$ to approximate the ReLU function, based on the work by Chabanne *et al.* exploring polynomial approximations of the ReLU function for HE applications [30]. A copy of each of the ciphertexts is made, each ciphertext is multiplied by itself (i.e., x^2), and the copy is added to the result (i.e., $+x$).

Applying weights in the first FC layer is similar to that in the Conv layer, but the weights are stored in plaintexts in a different order due to the difference in structures. Fig. 4 (right) shows one set of weights. After multiplications with the weight plaintexts and adding up, inter-slot addition is used to combine across slots, and the bias plaintexts are added to the ciphertexts. For expansion, a mask is used to retain only the first eight values in each ciphertext, and then those are each duplicated 128 times, filling the first 1,024 slots in each of the 16 output ciphertexts. By rotating ciphertexts by multiples of 1,024, the 16 outputs are combined into four ciphertexts with 4,096 slots filled.

Weights for the second FC layer are encoded into four plaintexts. Again, ciphertexts and plaintexts are multiplied, and the resulting ciphertexts are added up into a final ciphertext. Inter-slot addition is used with rotations of 1, 2, 4, 1024, and 2048, and the bias plaintext for this layer is added to the ciphertext. Expansion extracts the values in every eighth slot, up to slot 1016, using a mask and creates 16 copies using rotations of 1, 2, 4, and 1024.

The final weight plaintext is encoded, corresponding to the 12 labels, and applied. Inter-slot addition and addition with a bias plaintext complete the server-side computations. This final ciphertext is returned to the client. After decryption and decoding, the client chooses the label with the highest prediction score.

V. Evaluation

In this section, HEKWS is evaluated in terms of prediction accuracy, execution time, and other resource requirements.

TABLE II
HE Parameters for HEKWS

Polynomial degree N	Bit-length of q [33]	Maximum depth†
2^{14}	441	10

†{60, 30, 30, ..., 30, 30, 60} bits are allocated for 12 primes.

A. Experimental Setup

1) Dataset and Training: In 2018, Google released the Speech Commands Dataset [31] for use in training and testing KWS models. The dataset contains over 100,000 one-second audio samples split between 12 labels: 10 command words (up, down, yes, no, left, right, on, off, stop, go); no word; and unknown word. The dataset is partitioned into training, validation, and test sets with an approximate 85/10/5% split, respectively.

Our model was first implemented in Keras [32], then trained using the training and validation sets in the Speech Commands Dataset. With a batch size of 64, the model achieved a minimum validation loss after 32 epochs. The model was then modified, replacing the ReLU activation function with the linear approximation $x^2 + x$. The training was then resumed, before reaching the stopping criteria in 40 epochs.

2) HE Parameters: Our HEKWS is implemented using functions of the CKKS scheme in the Microsoft SEAL open-source library version 3.6.2 [28]. Table II shows the HE parameters used in HEKWS. There are mainly two parameters: polynomial degree N and bit-length of q (= $\log_2 q$). In general, $\log_2 q$ is proportional to N, and $\Delta\log_2 q/\Delta N$ decreases as a security level increases [12]. In fact, q is a product of multiple primes, and one prime is dropped whenever `HomMult` is executed. The first and last primes are used for special purposes in SEAL, so the number of primes minus 2 is the maximum available depth. There are ten stages in the CNN model of HEKWS where homomorphic multiplications are used (in order): normalization - masking - Conv layer - masking - approximate ReLU - FC1 layer - masking - FC2 layer - masking - softmax. Most of the homomorphic multiplications are `HomMultPlain`. Theoretically, `HomMultPlain` does not need to consume a depth every time. However, in SEAL, it consumes a depth for easy and safe use for beginners [19]. Therefore, HEKWS requires a depth of 10.

Our HEKWS targets a 128-bit security level, widely used in practical real-world applications. The execution time increases as the value of N increases, and therefore we set the N value to 2^{14}, which is the minimum value satisfying our target security level and depth. For our N value and security level, the SEAL manual suggests 441-bits for the $\log_2 q$ value [33]. To generate the maximum available depth of 10 with this bit number, 60 bits are allocated to the first and last primes, and 30 bits are allocated to the remaining 10 primes.

B. Prediction Accuracy

Table III compares the prediction accuracy of small-footprint KWS designs. To measure the prediction accuracy, we processed a random 1,000 keyword samples in Google's

TABLE III
PREDICTION ACCURACY COMPARISON

Non-HE-based Honk [5] w/o ReLU approx.	Non-HE-based Honk [5] w/ ReLU approx.	HEKWS
76.0%	72.1%	72.3%

TABLE IV
ERRORS IN A CASE WITH A DIFFERENT PREDICTION RESULT

Label	Prediction score Non-HE[†]	Prediction score HEKWS	Error (%)
0	5.746	5.829	1.45
1	4.373	4.444	1.62
<u>2</u>	12.720	**12.907**	1.47
3	3.727	3.787	1.61
4	0.330	0.341	3.13
5	-0.853	-0.856	0.33
6	6.938	7.051	1.64
7	-3.740	-3.787	1.25
8	-4.235	-4.290	1.30
9	7.502	7.624	1.63
10	-153.446	-155.744	1.50
11	**12.729**	12.889	1.26

[†]Non-HE-based Honk [5] w/ ReLU approximation

Speech Commands Dataset with an approximately even 8.3% split between labels. The accuracy of HEKWS was compared to that of the non-HE-based Honk [5]. In addition, the result of Honk with the approximate ReLU (i.e., $x^2 + x$) is shown in the second column to see the accuracy degradation in HE conversion.

Among the three small-footprint KWS designs, the Honk design without the ReLU approximation shows the highest accuracy of 76.0%. Note that this value is lower than the prediction accuracy of 77.9% presented in the Honk paper. This is because only the logarithm of the spectrogram was used, without the mel scaled spectrogram, in the training process of this experiment. Also, unlike the original Honk, noise was not added to the training set.

Using the ReLU approximation resulted in a 3.9% reduction in accuracy, as shown in the first and second columns. Besides this approximation, the homomorphic operations of the CKKS scheme generate errors in the results to preserve precision, which causes a slight change in the prediction score. However, HEKWS did not cause any significant degradation in prediction accuracy. Out of the 1,000 samples, only two samples were classified differently. Both were borderline cases, with the highest labels falling within a fraction of a percent. Table IV shows one such classification of a sample with label 2 (underlined) as the ground truth. The error is due to the HE conversion and is negligible for most samples, averaging 1.75%. Although HEKWS predicted the correct labels for those two samples, these are uncommon edge cases and do not indicate HEKWS outperforms the baseline.

C. Execution Time

Table V shows the average execution time per keyword of the proof-of-concept implementation of HEKWS, excluding the encoding, encryption, decryption, and decoding times. As hardware, a workstation with the Intel Xeon W-2295, 128GB RAM, and Ubuntu 18.04 LTS operating system was used. The first through third columns show the execution time in each homomorphic operation. Note that `HomAdd` and `HomMult` in this table include `HomAddPlain` and `HomMultPlain`, respectively, and the second column includes the execution time in `Relin` and `Rescale` as well.

TABLE V
EXECUTION TIME PER KEYWORD (MILLISECONDS)

`HomAdd`[†]	`HomMult`[‡]`+Relin+Rescale`	`HomRot`	Total
185	3,556	14,805	18,546

[†]Including `HomAddPlain`
[‡]Including `HomMultPlain`

TABLE VI
RPI 4 CLIENT-SIDE EXECUTION TIME (SECONDS)

Computation		Time
Key generation (one-time)	Secret key	0.02
	Public key	0.30
	Relin key	3.33
	Galois key	99
Encoding + Encryption		1.01
Decryption + Decoding		0.08

In total, it takes 19 seconds to process a single encrypted keyword. For reference, the non-HE version had an execution time of 64 milliseconds on the same machine. The execution time for homomorphic additions accounts for 1% of our total execution time. Homomorphic multiplications account for 19%, with `HomMult+Relin`, `HomMultPlain`, and `Rescale` using 2%, 3%, and 14%, respectively. Each `HomMultPlain` requires shorter execution time compared to each `HomMult+Relin`. However, the number of `HomMultPlain` is larger than that of `HomMult+Relin`, so `HomMultPlain` takes up a larger percentage of the execution time. The remaining 80% is from `HomRot`. This is a relatively long time, but allows for the use of a single input ciphertext, significantly reducing client workload and network transfers.

As shown in Table VI, running the key generation function on a Raspberry Pi (RPi) 4 Model B with a 4GB RAM took 102 seconds, with 99 seconds of that spent generating Galois keys. Since the key generation is usually a one-time process and the generated keys are reused, this process is not a big burden on the client-side. Encoding and encryption are done in about one second, and decryption and decoding use additional 84 milliseconds.

D. Client Resources Requirements

Each classification requires the client to send a 1.7MB ciphertext. Table VII shows the reduction in ciphertext transmission requirements using the proposed packing technique compared to the naïve and expanded packing techniques described in Section III-A. In the case of key transmission, the first time the client establishes a connection with the server, it makes a one-time transfer of the relinearization and Galois keys. Once received, the server can store and use the keys indefinitely, as mentioned in the previous subsection. The

TABLE VII
THE NUMBER OF INPUT CIPHERTEXTS (SIZE) PER KEYWORD

Naïve packing	Expanded packing	Proposed packing
1280 (2.4GB)	32 (54.4MB)	1 (1.7MB)

public and secret keys used for encryption and decryption are stored solely on the client and have sizes of several MB.

VI. CONCLUSIONS

Our HEKWS with the novel packing technique accepts and returns single ciphertexts for secure classification of a potential command word. In the proposed packing technique, a single ciphertext is expanded on a server to effectively apply different weights. It requires masking and rotations but helps shift the burden from a resource- and power-hungry client to a server. Although there is a decrease in accuracy due to the ReLU approximation, there is a negligible decrease due to the homomorphic operations. To the best of the authors' knowledge, this is the first paper that applies a HE scheme to KWS in CPS and IoT.

The 19-second processing time of our proof-of-concept implementation can be reduced with parallelization. For example, the homomorphic multiplications in the Conv layer can be performed in parallel. This is true for most of the multiplications and additions, and many of the rotations in HEKWS. Furthermore, the polynomial degree of 2^{14} used in this paper can be reduced by using other open-source HE libraries where `HomMultPlain` does not consume a depth every time. Custom hardware accelerators can be used as well to improve the speed. We envision that these optimizations would promote real-world use of HEKWS by improving the speed. Besides the speedup, the scalability of the number of keywords and the number of layers of CNN can be an interesting research topic.

ACKNOWLEDGMENT

We thank Dr. Rob Rutenbar, Dr. Jung Hee Cheon, Dr. Ha-Kyung Kong, Wonhee Cho, and Keewoo Lee for their continued support.

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2020-0-00840, Development and Library Implementation of Fully Homomorphic Machine Learning Algorithms supporting Neural Network Learning over Encrypted Data). In addition, this material is based upon work supported by the National Science Foundation under Grant No. 2105373.

REFERENCES

[1] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*, Berkeley, CA, USA:Lee & Seshia, 2011.

[2] "Why It's Critical to Keep IoT Security Top-of-Mind," [Online]. Available: https://www.aeris.com/news/post/why-its-critical-to-keep-iot-security-top-of-mind

[3] G. Chen, C. Parada, and G. Heigold, "Small-footprint Keyword Spotting Using Deep Neural Networks," in *Proc. ICASSP*, 2014.

[4] R. Tang and J. Lin, "Deep Residual Learning for Small-footprint Keyword Spotting," in *Proc. ICASSP*, 2018.

[5] R. Tang and J. Lin, "Honk: A Pytorch Reimplementation of Convolutional Neural Networks for Keyword Spotting," *arXiv preprint arXiv:1710.06554*, 2017.

[6] T. N. Sainath and C. Parada, "Convolutional Neural Networks for Small-footprint Keyword Spotting", in *Proc. INTERSPEECH*, 2015.

[7] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello Edge: Keyword Spotting on Microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.

[8] R. C. Rose and D. B. Paul, "A Hidden Markov Model based Keyword Recognition System," in *Proc. ICASSP*, 1990.

[9] M. H. Cohen, M. H. Cohen, J. P. Giangola, and J. Balogh, *Voice User Interface Design*, Addison-Wesley, 2004.

[10] M. Vacher, F. Aman, S. Rossato, F. Portet, and B. Lecouteux, "Making Emergency Calls More Accessible to Older Adults Through a Hands-free Speech Interface in the House," *ACM Trans. Access. Comput.*, vol. 12, no. 2, pp. 1-25, Jun. 2019.

[11] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Proc. ASIACRYPT*, 2017.

[12] J. H. Cheon, A. Costache, R. C. Moreno, W. Dai, N. Gama, M. Georgieva, S. Halevi, M. Kim, S. Kim, K. Laine, Y. Polyakov, and Y. Song, "Introduction to Homomorphic Encryption and Schemes," in *Protecting Privacy through Homomorphic Encryption*, Springer, 2021. pp. 3-28.

[13] Z. Brakerski, "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP," in *Proc. CRYPTO*, 2012.

[14] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," *Cryptology ePrint Archive*, Report 2012/144, 2012.

[15] L. Ducas and D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second," in *Proc. EUROCRYPT*, 2015.

[16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34-91, 2020.

[17] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[18] S. Kannivelu and S. Kim, "A Homomorphic Encryption-based Adaptive Image Filter Using Division Over Encrypted Data," in *Proc. RTCSA*, 2021.

[19] M. Cho, K. Lee, and S. Kim, "HELPSE: Homomorphic Encryption-based Lightweight Password Strength Estimation in a Virtual Keyboard System," in *Proc. GLSVLSI*, 2022.

[20] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme," in *Proc. FCCM*, 2020.

[21] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "FPGA-based Accelerators of Fully Pipelined Modular Multipliers for Homomorphic Encryption," in *Proc. ReConFig*, 2019.

[22] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *Proc. ICML*, 2016.

[23] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure Outsourced Matrix Computation and Application to Neural Networks," in *Proc. CCS*, 2018.

[24] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A Full RNS Variant of Approximate Homomorphic Encryption," in *Proc. SAC*, 2018.

[25] Z. Brakerski, C. Gentry, and S. Halevi, "Packed Ciphertexts in LWE-based Homomorphic Encryption," in *Proc. PKC*, 2013.

[26] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in *Proc. USENIX Security*, 2018.

[27] M. Abadi *et al.*, "Tensorflow: A System for Large-scale Machine Learning," in *Proc. OSDI*, 2016.

[28] "Microsoft SEAL (release 3.6)," [Online]. Available: https://github.com/Microsoft/SEAL

[29] "PALISADE Lattice Cryptography Library (release 1.11.5)," [Online]. Available: https://palisade-crypto.org/

[30] H. Chabanne, A. De Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving Classification on Deep Neural Network," *Cryptology ePrint Archive*, Paper 2017/035, 2017.

[31] P. Warden, "Speech Commands: A Dataset for Limited-vocabulary Speech Recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[32] A. Gulli and S. Pal, *Deep learning with Keras*, Packt Publishing Ltd, 2017.

[33] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine, "Simple Encrypted Arithmetic Library v2.3.0," [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/12/sealmanual.pdf