DECODING HOMOMORPHICALLY ENCRYPTED FLAC AUDIO WITHOUT DECRYPTION

Yuanyuan Tang[1], *Bin Zhu*[3], *Xiaojing Ma*[1,2*], *P. Takis Mathiopoulos*[4], *Xia Xie*[1], *and Hong Huang*[1]

[1]National Engineering Research Center for Big Data Technology and System
Service Computing Technology and System Lab
Big Data Security Engineering Research Center
School of Computer Science and Technology
Huazhong University of Science and Technology
[2]Shenzhen Huazhong University of Science and Technology Research Institute
[3]Microsoft Research Asia
[4]Department of Informatics and Telecommunications, University of Athens

ABSTRACT

Homomorphic Encryption (HE) allows processing ciphertext data, but it is a challenge to enable complex methods such as multimedia decompression in the HE domain. In this paper, we propose a novel scheme to enable FLAC (Free Lossless Audio Codec) decompression in the HE domain. FLAC applies linear prediction to predict the current sample and Golomb coding to encode residuals. FLAC decoding relies heavily on dynamic controls that HE does not support due to unknown values of control variables after encryption. Our scheme regularizes dynamic controls in FLAC decoding with static controls by calculating an encrypted matching bit for each possible value of a control variable and producing candidate results as if it were a match. The summation of each possible value's candidate results multiplied by its matching bit is equivalent to selecting the results of the matched control value. Our FLAC decoding scheme enables Single-Instruction Multiple-Data (SIMD): multiple (e.g., 256) plaintexts are packed and encrypted into a single ciphertext, and decoding one encrypted frame corresponds to decoding multiple plaintext frames. Our scheme is applicable to other audio compression standards based on similar technologies. Experimental results are also reported.

Index Terms— FLAC, privacy-preserving processing, homomorphic encryption.

1. INTRODUCTION

A growing amount of data has been uploaded to public clouds. Losing control of their data is an important concern for virtually all cloud users. To address this issue, privacy-preserving data processing has been actively studied in recent years. Towards this goal, Homomorphic Encryption (HE) [1] allows computations on ciphertexts to produce an encrypted version of the desirable result. This would allow clouds to conduct required data processing in the HE domain while the privacy of the data is fully preserved.

A great effort has been directed to enable commonly used algorithms and methods in the HE domain, such as simple statistical functions [2], discrete cosine transform [3], bubble sort [4], and AES [5]. It is a great challenge to enable complex algorithms and methods in the HE domain since HE does not support any dynamic control structure due to the fact that a control value is encrypted and thus unknown. This is particularly true for multimedia decompression, which relies heavily on dynamic controls in decoding a bitstream. We have introduced recently a novel approach to enable JPEG decompression in the HE domain [6], which regularizes Huffman decoding with static controls to output only one DCT coefficient in each iteration of decoding. This is the first decompression enabled in the HE domain.

In this paper, we enable the second decompression, FLAC [7], in the HE domain. FLAC is the most widely supported lossless audio compression standard. Unlike JPEG that relies on Huffman coding, FLAC relies on linear prediction and Golomb coding [8], which are widely used in other compression standards such as SILK [9], Shorten [10], Apple Lossless, and MPEG-4 Audio Lossless Coding [11]. Like Huffman decoding in JPEG in [6], FLAC decoding relies heavily on dynamic controls. The main challenge is to regularize the dynamic controls in FLAC decoding with static controls and sequential operations. To tackle this challenge, we examine each possibility for an encrypted control value by computing an encrypted matching bit that indicates if the possibility matches the control value or not, and would produce candidate results (outputs and new bitstreams, as describing in de-

*Corresponding author: Xiaojing Ma (lindahust@hust.edu.cn). This work was supported in part by National Natural Science Foundation of China (61771211), Fundamental Research Funds for the Central Universities (2017KFYXJJ064), Shenzhen Fundamental Research Program (JCYJ20170413114215614) and Hubei Provincial Natural Science Foundation General Program (2018CFB200).

tail in Section 3.2) as if it were a match. Candidate results from different possibilities are of the same form, and the summation of each result multiplied by its encrypted matching bit over all possibilities is equivalent to selecting the result of the matched one in decoding plaintext FLAC audio.

There is a side benefit in our regularized FLAC decoding: our decoding scheme enables Single-Instruction Multiple-Data) (SIMD) [12] packing and decoding: multiple (e.g., 256) plaintexts are packed and encrypted into a single ciphertext and decoding one ciphertext frame corresponds to decoding multiple plaintext frames. This SIMD decoding is unimaginable for a conventional FLAC decoding procedure.

2. FLAC OVERVIEW

FLAC [7] uses linear prediction to predict the current sample from preceding samples and Golomb coding to encode the residual. A FLAC bitstream comprises a file header, metadata blocks, and frames. A frame comprises a frame header and subframes, one subframe per channel. FLAC supports 4 methods of prediction. We choose the fixed linear predictor in our studies and describe it in this section.

During encoding, audio samples are partitioned into blocks, each with $blocksize$ samples per channel. The channels in a block may be combined, such as converted to mid channel of bps bits and side channel of $bps + 1$ bits for bps-bit stereo audio. Each channel is then encoded independently into a subframe using a linear predictor of a fixed $order \in [0, 4]$ with *order* preceding samples to predict the current sample [10, 13] and Golomb coding [8] to encode the residual. The initial *order* samples for prediction, called *warm-up samples*, are written into a subframe without coding.

In Golomb coding, an integer n is first encoded to $N \geq 0$ as follows: $N = 2n$ if $n \geq 0$ and $N = 1 - 2(n + 1)$ if $n < 0$. N is then split into two parts using a Rice parameter R: MSB as the quotient N div 2^R and LSB as the remainder N mod 2^R. MSB is encoded with unary coding while LSB is in binary representation with R bits. Unary coding encodes an integer $t \geq 0$ with t zeros followed by a stop bit one. For example, if the Rice parameter $R = 3$ and $n = 18$, then $N = 36$, and $MSB = 36/2^3 = 4$ encoded as 00001, $LSB = 36\%2^3 = 4$ encoded as 100. The resulting bits are 00001100. The encoder writes the Rice parameter and coded residuals into the residual block of a subframe.

In decoding a subframe, *order* is obtained from the subframe header, and *order* warm-up samples are extracted. To decode each of $(blocksize - order)$ residuals, the Rice parameter R is obtained from the residual block, MSB is obtained by counting bit 0 before a stop bit 1, and LSB is the following R bits, resulting in $N = (MSB \times 2^R + LSB)$. If N is even, then the residual $n = N/2$. If N is odd, then the residual $n = (1 - N)/2 - 1$. After decoding $(blocksize - order)$ residuals, the $(blocksize - order)$ samples in the subframe can be recovered by an inverse process of linear prediction, and the decoded subframes can be recombined if needed to recover the samples of the original channels.

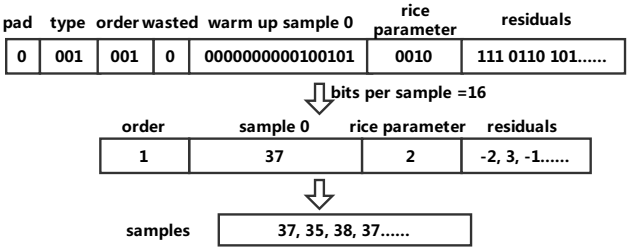

Fig. 1. An example of decoding a subframe using fixed linear prediction.

Fig. 1 shows an example of decoding a subframe using fixed linear prediction with 16 bits per sample, within $type = 001$ represents using fixed linear prediction and $order = 1$ indicates one warm-up sample. The warm-up sample is obtained from the subsequent 16 bits, which is 37. From the figure, the Rice parameter $R = 2$. Residuals are recovered, $residual_0 = -2$, $residual_1 = 3$, $residual_2 = -1, \cdots$. With $order = 1$, the current sample can be recovered by adding its decoded residual to the preceding sample, resulting in decoded samples: 37, 35, 38, 37, $\cdots$, where the first integer is the warm-up sample.

3. FLAC DECODING IN THE HE DOMAIN

3.1. Encryption

Like other full multimedia encryption, we encrypt FLAC payloads while leave headers unencrypted. More specifically, the FLAC file header, metadata blocks, and the frame header are unencrypted, while subframes (including subframe headers) in a frame are encrypted with plaintexts $\in Z_2$, wherein addition and multiplication are equivalent to binary XOR and AND operations, respectively. For $b \in Z_2$, its ciphertext is denoted as $[b]$. For an integer n, $[n]$ denotes its binary encryption, i.e., encryption of each bit in its binary representation. In our work, BGV [14] is used as the homomorphic encryption.

The unencrypted portion (i.e., the file and frame headers and metadata) can be used for identifying the basic audio and compression information and for finding specific frames. Since the actual content is contained in subframes, which is fully encrypted, adversaries cannot access any content information except the length of each frame.

As we will see next, each frame is decoded with the same sequential operations. This allows us to apply SIMD to pack D plaintext frames into a single ciphertext frame, where D is determined by the parameters of homomorphic encryption. Decoding one ciphertext frame is equivalent to decoding D plaintext frames.

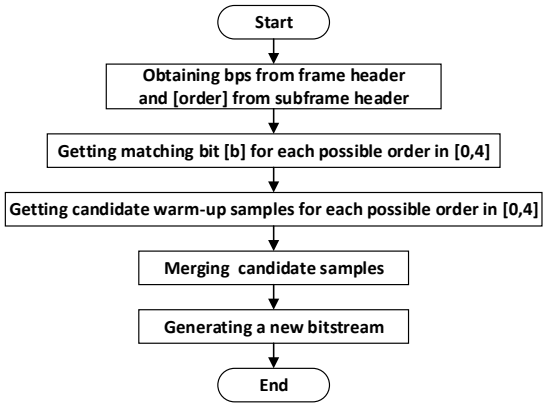

Fig. 2. Flow chart of extracting warm-up samples in the homomorphic encryption domain

3.2. Decoding a subframe in the HE domain

Decoding parameters in the subframe such as $order$ and Rice parameter R are encrypted and thus unknown. Without knowing $order$, we don't know how many warm-up samples are used or where the bits of a residue are located. Without knowing Rice parameter R, we don't know how many bits for LSB. In addition, unary decoding cannot be executed in the HE domain since a stop bit 1 cannot be identified, leading to undetermined MSB. These challenges are tackled by comparing each possibility with the bitstream to produce an encrypted matching bit, multiplying the matching bit with the result assuming the possibility is a match, and then taking a summation of their products over all possibilities. The obtained result is equivalent to selecting the result of the matched possibility. The number of channels is in the unencrypted metadata and thus the number of subframes in a frame is known. Subframes in a frame are decoded sequentially. We focus on describing decoding one subframe in this section.

3.2.1. Extract warm-up samples

The prediction order, $order \in [0, 4]$, is represented by 3 bits in the encrypted subframe header, denoted as $[order]$, which determines the number of warm-up samples. For each possible value $k \in [0, 4]$ of $order$, we compare it with $[order]$ to produce an encrypted matching bit $[b_k]$, and obtain its warm-up samples and drop consumed bits as if k is a match. The summation of a result, either warm-up samples or remaining bits, multiplied by its matching bit over all possible values of k is equivalent to selecting the warm-up samples or remaining bits of the matched $k = order$. Fig. 2 shows the workflow of the process.

Algorithm 1 shows the detail of producing a matching bit $[b]_k$ for a candidate value k of $order$. It multiples a bit of $[order]$ if the corresponding bit of the candidate is 1 or its inverse otherwise. In the algorithm, $[x] + [1]$ is equivalent to the inverse ([NOT x]) of $[x]$ for a bit x. Among all candidates, only the matching candidate has its $[b]_{order} = [1]$. All other candidates have their $[b]_{k \neq order} = [0]$.

Algorithm 1: Matching prediction order

Input: Encrypted prediction order, $[order]$, of 3 bits $\{[order]_i\}$ from bitstream, and k of 3 bits $\{k_i\}$ as a candidate value of $order$.
Output: Encrypted matching bit $[b]_k$.

```
[b]_k = [1];
for i = 0 to 2 do
    if k_i = 1; then
        [b]_k = [b]_k * [order]_i;
    else
        [b]_k = [b]_k * ([order]_i + [1]);
    end
end
```

For each $order$ candidate k, extracting its k candidate warm-up samples $\{[sample_j]^k \mid j \in [0, k-1]\}$ is straightforward after getting bps from the unencrypted frame header. The final warm-up samples can be found as follows.

$$[Sample_j] = \sum_{k, j<k} [sample_j]^k * [b]_k, j = 0, 1, ..., 3.$$

For each order candidate k, a new bitstream $[Newbits]^k$ is generated by dropping its consumed bits. These candidates $\{[Newbits]^k\}$ are merged to get a new bitstream for the next decoding:

$$[Newbits] = \sum_{k=0,\cdots,4} [Newbits]^k * [b_k].$$

3.2.2. Extract residuals

To get MSB for a residual, we need to search for the first stop bit 1. In our scheme, the maximum value of MSB (MAX_MSB) is computed and written into a padding metadata block during encryption. At decoding, MAX_MSB is extracted from the unencrypted metadata block. Similar to extracting warm-up samples in Section 3.2.1, for each candidate $msb \in [0, MAX\_MSB]$ of MSB, we calculate a matching bit according to Algorithm 2 and merge these candidates to obtain the final $[MSB]$ and produce an updated bitstream $[Newbits]$ in the same way as in Section 3.2.1.

LSB of a residual depends on encrypted Rice parameter $[R]$. In a similar manner, for each candidate $r \in [0, 14]$ of R, we calculate an encrypted matching bit $[b]$, obtain its candidate LSB, and update its MSB. For example, if $r = 3$, then its LSB is the first 3 bits in the bitstream, $[MSB]$ is updated to $([MSB] \ll 3)$. A candidate residual is then computed by

Algorithm 2: Matching MSB

Input: A candidate msb of MSB and an encrypted bitstream $[bits]$ ($[bits]_i$ is i-th bit of $[bits]$)
Output: Encrypted matching bit $[b]$ for candidate msb.
$[b] = [1]$;
for $i = 0$ *to* msb **do**
    $[b] = [b] * ([bits]_i + [1])$;
end
$[b] = [b] * [bits]_{i+1}$;

the following equations:

$$[Residual] = ([LSB] + [MSB] \ll R) \gg 1 \quad (1)$$

$$[Residual]_i = [Residual]_i \oplus [S] \quad (2)$$

$$[Residual] = [S][Residual] \quad (3)$$

where $[S]$ is last bit of LSB, $[Residaul]_i$ is the i-th bit of $[Residual]$. In Eq. 3, $[S]$ is inserted into $[Residual]$ as its sign bit. In the above equations, "$\ll$" is left shift, "$\gg$" is right shift, "$+$" means addition with carry, and "$\oplus$" means bitwise addition without carry. The candidate residual and the new bitstream for each candidate r of R are then merged respectively in a similar manner as in Section 3.2.1 to produce the final residual and new bitstream.

To get all residuals, a set of candidate residuals is generated for each possible prediction order, and these candidate residual sets are merged to get the final $blocksize$-$order$ residuals.

3.2.3. Extract all samples

With warm-up samples obtained in Section 3.2.1 and prediction residuals obtained in Section 3.2.2, all the samples can be recovered with linear operations. This process is repeated until all subframes in a frame are decoded.

4. EXPERIMENTAL RESULTS

We have implemented the proposed scheme in C++ based on HELib [15] that implements BGV, and the independent FLAC Group's implementation of FLAC [7]. The experiments were carried out on a server with Intel Xeon CPU E5-2680 v4 of 2.40GHz with 14 physical (28 logical) cores and 256GB of RAM running CentOS Linux release 6.5. All test audio clips were of one-second audio with 44.1kHz, 2 channels, and 16 bits per sample, and $blocksize$ was set to 18, resulting in 2450 plaintext frames per audio clip. The following BGV parameters were chosen in our experiments: $m = 4369$, $p = 2$, $d = 16$, $r = 1$, and levels $L = 41$, which led to $D = 256$, i.e., 256 plaintext frames were packed together and encrypted into one ciphertext frame, resulting in 10 (=$\lceil \frac{2450}{256} \rceil$) encrypted frames. After decoding in the HE domain, we decrypted the result and compared with the decoding result by the FLAC decoder [7] without encryption to ensure correctness of our results.

Table 1. Running time (s) of decoding one frame.

FLAC File	Size	MAX_MSB	Time
romantic_f0_fr0	232	14	2782.52
romantic_f0_fr1	320	14	3366.17
romantic_f0_fr2	336	14	3454.33
romantic_f0_fr3	352	14	3569.17
romantic_f0_fr4	376	14	3732.64
romantic_f1_fr0	336	16	3478.3
romantic_f1_fr1	376	16	3747.68
rock_fr0	376	14	3754.46
rock_fr1	440	14	4236.99
absolute_fr0	232	17	2769.07
absolute_fr1	336	17	3451.78

Table 1 shows the running time of decoding one frame with different files and frames. We can observe that the decoding time of frames with similar lengths is similar in value. This is because the bitstream's length determines the number of multiplications in generating a new bitstream and the number of bits that need bootstrapping, which dominates the execution time. MAX_MSB does not show much impact on the decoding time because MAX_MSBs in Table 1 are all less than the number of logical cores available in the experimental computer, and thus they could be executed in parallel.

Like other algorithms and methods executed in the HE domain, it is still too slow to decode FLAC audio in the HE domain, which makes the proposed scheme impractical for most applications. This issue should be lessened with the progress of homomorphic encryption. A recent paper proposed a fast homomorphic encryption scheme that can perform bootstrapping in less than 0.1s [16], much faster than HELib's bootstrapping, which takes about 75s on our test computer with the same BGV parameters used in our experiments. This may bring a hope to apply the proposed FLAC decoding in the homomorphic encryption domain to some real applications. We plan to test our proposed scheme with this fast HE scheme once its implementation supports SIMD.

5. CONCLUSIONS

In this paper, we proposed a novel scheme to enable FLAC decompression in the homomorphic encryption domain by regularizing dynamic controls in FLAC decoding with static controls. Our FLAC decoding enables SIMD: multiple (e.g., 256) plaintexts are encrypted into one ciphertext, and decoding one encrypted frame corresponds to decoding multiple plaintext frames. Our scheme is applicable to other audio compression standards based on linear prediction and Golomb coding such as SILK, Shorten, Apple Lossless, and MPEG-4 ALS.

6. REFERENCES

[1] Craig Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA*, 2009, pp. 169–178.

[2] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan, "Can Homomorphic Encryption Be Practical?," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, New York, NY, USA, 2011, CCSW '11, pp. 113–124.

[3] Tiziano Bianchi, Alessandro Piva, and Mauro Barni, "Encrypted Domain DCT Based on Homomorphic Cryptosystems," *EURASIP Journal on Information Security*, vol. 2009, no. 1, pp. 1–12, 2009.

[4] Carlos Aguilar-Melchor, Simon Fau, Caroline Fontaine, Guy Gogniat, and Renaud Sirdey, "Recent Advances in Homomorphic Encryption: A Possible Future for Signal Processing in the Encrypted Domain," *IEEE Signal Processing Magazine*, vol. 30, no. 2, pp. 108–117, Mar. 2013.

[5] Craig Gentry, Shai Halevi, and Nigel P. Smart, "Homomorphic Evaluation of the AES Circuit," in *Proceedings of Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA*, 2012, pp. 850–867.

[6] Xiaojing Ma, Changming Liu, Sixing Cao, and Bin Zhu, "JPEG Decompression in the Homomorphic Encryption Domain," in *Proceedings of ACM Multimedia Conference on Multimedia Conference, MM 2018, Seoul, Korea*, 2018, pp. 905–913.

[7] Josh Coalson and Xiph.Org Foundation, "FLAC: Free lossless audio codec," `https://xiph.org/flac/`, 2017.

[8] Solomon Golomb, "Run-length encodings (Corresp.)," *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, 1966.

[9] Koen Vos, Soeren Jensen, and Karsten Soerensen, "SILK speech codec - IETF draft," `https://tools.ietf.org/html/draft-vos-silk-02`, 2010.

[10] Tony Robinson, "SHORTEN: Simple lossless and near-lossless waveform compression," `http://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/robinson_tr156.pdf`, 1994.

[11] Tilman Liebchen, "An introduction to MPEG-4 audio lossless coding," in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2004, Montreal, Quebec, Canada*, 2004, pp. 1012–1015.

[12] Nigel P. Smart and Frederik Vercauteren, "Fully homomorphic SIMD operations," *Designs, Codes and Cryptography*, vol. 71, no. 1, pp. 57–81, Apr. 2014.

[13] Mat Hans and Ronald W Schafer, "Lossless compression of digital audio," *IEEE Signal Processing Magazine*, vol. 18, no. 4, pp. 21–32, 2001.

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," *ACM Transactions on Computation Theory*, vol. 6, no. 3, pp. 13:1–13:36, 2014.

[15] Shai Halevi, "Helib: An implementation of homomorphic encryption," `https://github.com/shaih/HElib`, 2013.

[16] Léo Ducas and Daniele Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, Proceedings, Part I*, 2015, pp. 617–640.