LLM-MARK: A Computing Framework on Efficient Watermarking of Large Language Models for Authentic Use of Generative AI at Local Devices

Shiyu Guo, Yuhao Ju, Xi Chen, Jie Gu
Northwestern University, Evanston, IL
{ ShiyuGuo2021, YuhaoJu2017, XiChen2020}@u.northwestern.edu, jgu@northwestern.edu

***Abstract*—As generative AI such as ChatGPT rapidly evolves, the increasing incidence of data misconduct such as the proliferation of counterfeit news or unauthorized use of Large Language Models (LLMs) presents a significant challenge for consumers to obtain authentic information. While new watermarking schemes are recently being proposed to protect the intellectual property (IP) of LLM, the computation cost is unfortunately too high for the targeted real-time execution on local devices. In this work, a specialized hardware-efficient watermarking computing framework is proposed enabling model authentication at local devices. By employing the proposed hardware hashing for fast lookup and pruned bitonic sorting network acceleration, the developed architecture framework enables fast and efficient watermarking of LLM on the small local devices. The proposed architecture is evaluated on Xilinx XCZU15EG FPGA, demonstrating 30x computing speed-up, making this architecture highly suitable for integration into local mobile devices. The proposed algorithm to architecture codesign framework offers a practical solution to the immediate challenges posed by LLM misuse, providing a feasible hardware solution for Intellectual Property protection in the era of generative AI.**

I. INTRODUCTION

The recent advancements in generative AI enable high-quality text generations that are difficult to distinguish between human or machine origins. While the generative AI offers powerful new capabilities for human assistance such as customer support, content creation, essay composition, education and tutoring, etc., the recent explosive use of such technology calls for an urgent action for implementing security measures on the large language models (LLM). Malicious usage of LLM such as plagiarism in academic activities, forged generations of news or articles, unauthorized use of copyrighted models poses significant threats to the integrity of information contents in social media, education channels and daily communications. As one of the recent announcements, major AI companies, such as Open AI, Googles, have pledged to provide watermark techniques for the AI-generated text contents.

Despite the significant efforts being poured into the authentication of the generated contents of LLM, a viable watermarking technique for LLM is still under development. Although a deep neural network (DNN) watermarking framework was proposed in [1], it cannot be directly used for language models. For LLM models, two types of watermarking techniques are commonly used. The first type of techniques uses "Post-hoc Detectors" which incorporates heuristics or model-based characteristics to detect the differences between natural languages from human being and the machine-generated models. This method does not alter the outputs of the LLM models for detection. For example, the GLTR [2,3] detector applies the solution based on statistical methods that can detect generation artifacts to check whether the text is generated by a model. The AI classifier [4] is a detector from OpenAI to distinguish AI-generated text by detecting false claims through a trained model. The method is not fully reliable due to the limitations on short text and the potential misleading human-written text for evading the detection. Another model based detector, DetectGPT [5], uses only log probabilities computed by the model and random perturbations of the passage from another generic pre-trained model instead of separate training procedure and extra datasets. As the LLM is being rapidly developed to behave indistinguishable to the human natural speeches, the post-hoc approaches face difficulty in achieving detection accuracy. A second class of the watermarking techniques, as recently being proposed by [6], employs the use of LLM to influence the content being generated so that the generated text follows certain characteristics that can be detected by the users holding the keys which are used in the generation process. This type of watermarking techniques requires sophisticated changes or appendages to the generated texts with a balance between detectability and quality of the text. The benefits of this technique are as follows: (1) the detection can be performed without knowledge of the model parameters, (2) no retraining is needed to the LLM, (3) watermarks can be detected by only a portion of the text, (4) confidence score is available to quantify the watermarking results, (5) watermarks cannot be removed or evaded without significant modifications. As a result, this technique represents the state-of-the-art watermarking method for the current LLM. As an example shown in Fig. 1 from the watermarking method in [6], by embedding the watermarking pattern in the text, the watermarking can be detected while remaining indiscernible to humans.

Input Text	No Watermark	With Watermark	Z-Score No Watermark	Z-Score w/ Watermark
[...]Then the study team assessed the response of two ophthalmologists who viewed and rated the pictures on a desktop PC and did the same on an iPhone. \n The results of the study, published in the Archives of Ophthalmology, show that the doctors found the iPhone images good or better than desktop images and rated them high. \n While one	iphone was better than the other, the iPhone images were better than desktop images. \n A common misconception with ophthalmologists, they say, is that the images should be viewed in a progressive way. \n However, the study suggests that the images may be viewed in a way that is more progressive in nature, and that there may be better responses to the images.[...]	iphone was able to get the highest resolution images of the inner eye to be seen by the other, the researchers found that the pictures were not as good as the actual picture. \n The researchers found that the image quality was improved by using smartphone apps such as DoF (de-facto FOV) and DoF (de-facto FOV).[...]	-0.46	11.45

Fig. 1. Watermarking of LLM example [6].

While the watermarking method in [6] offers a viable direction for authentication of the LLM models, the computation cost for generation of watermarks is very high. As will be shown in Section 2, the vocabulary lookup and entropy sorting operations cause even more runtime than the LLM model by itself on a high-end processor. As many applications for authentic use of LLMs require local execution of LLM model as protected IPs, running watermarking tasks at local devices suffer from the limited computing resources offered by mobile processors or low-cost dedicated silicon solutions such as small FPGAs. To overcome this challenge, this paper proposes a computing framework LLM-MARK for watermarking LLM models specially for resource-constrained local devices. The contributions of this paper include: (1). It is the first algorithm and architecture codesign of the LLM watermarking for generative AI model authentication and IP protection. (2). Special Toeplitz hash function is developed for both algorithm and hardware to accelerate the lookup procedure and green list generation by 243x for watermarking and detection. (3). The pruned bitonic soring network is implemented to reduce the latency of large vocabulary logit vector soring by 34.7x. (4) End-to-end evaluation on Xilinx XCZU15EG FPGA shows total 30x run cycle reduction for watermarking LLM results and 22.8x run cycle reduction for watermark detection.

II. Baseline Watermarking Algorithm for LLMs

Fig. 2 (a) shows the data flow of the watermarking process [6]. The prompt text is sent to LLM for next-token prediction. During each iteration, there will be a prediction logit vector from the transformer model representing the possibility for the next token. The watermark processor generates a random "green list" for each token. The possible following tokens are checked against the generated green list. The word in the green list is given higher possibility to be used for the next token. This process manipulates the probability distribution of the logic vector and sampling the next token based on the watermarked distribution. As a result, the generated text will be biased toward the green list. The watermark detection is based on the statistical probability of the green list appearance in the result. To control the text quality, bias is chosen so that only tokens with lower impact to text quality will be replaced by a green list tokens.

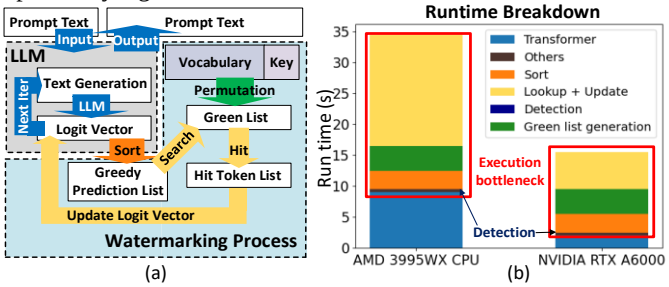


Fig. 2. (a) Processing flow of watermarking method. (b) Runtime breakdown of the watermarking operation in a microprocessor.

Algorithm 1 shows more details of the algorithm. Firstly, by providing a sequence of prompt text to the LLM, an output logit vector for the whole vocabulary list |V| is generated for each word. Next, by using user-defined permutation scheme, a "green list" with size of $\gamma \times |V|$ is generated. Third, the processor will sort the logit vector and check N most possible token from the sort result, for all the "green list" logit, add δ to the logit vector, and proceed to the next iteration.

Since the watermarked distribution is generated without knowledge of the green list bias, z score for watermarking detection can be defined as:

$$z = \frac{2(|S|_G - T/2)}{\sqrt{T}} \quad (1)$$

$|S|_G$ is the number of green list tokens. T is the number of tokens. The text is considered as watermarked if the z score passes a certain threshold.

To evaluate the hardware costs of the technique, we run the algorithm on local personal devices, i.e. AMD 3995WX CPU with 128 threads and NVIDIA RTX A6000 with a small LLM model OPT-125M [7] as used in the original method. Fig. 2 (b) shows the runtime breakdown. For example, a vocabulary size of 50272 is used in the experiment. The green list takes a fraction of the vocabulary list, e.g. 10%~40%, which is a large array for the watermarking process. It is observable the watermarking operation exceeds the runtime of the transformer model which is used to generate the text. The primary computation tasks for the watermarking process include greedy prediction lookup, sorting and green list generation, generating significant runtime overhead to the LLM model. In this work, an acceleration architecture is proposed and implemented in hardware to reduce the major execution overhead in end-to-end watermarking applications with a special of focus on the three major computing tasks, i.e. lookup, sorting and green list generation.

Algorithm 1 Watermarking Algorithm with Input Prompt

Require: prompt, $s_{(-N_p)}, \ldots, s_{(-1)}$
green list size, $\gamma \in (0, 1)$
hardness parameter, $\delta > 0$

1: **for** $t = 0, 1, \ldots$ **do**
2: Apply the language model to prior tokens $s_{(-N_p)}, \ldots, s_{(t-1)}$ to get a logit vector $\mathbf{l}^{(t)}$ over the vocabulary.
3: Compute a hash of token $s^{(t-1)}$, and use it to seed a random number generator.
4: Using this random number generator, randomly partition the vocabulary into a "green list" G of size $\lceil \gamma |V| \rceil$, and a "red list" R of size $\lfloor (1-\gamma)|V| \rfloor$.
5: Add δ to each green list logit. Apply the softmax operator to these modified logits to get a probability distribution over the vocabulary.
6: $$\hat{p}_k^{(t)} = \begin{cases} \frac{\exp(\mathbf{l}_k^{(t)}+\delta)}{\sum_{i\in G}\exp(\mathbf{l}_i^{(t)}+\delta)+\sum_{i\in R}\exp(\mathbf{l}_i^{(t)})}, & k \in G \\ \frac{\exp(\mathbf{l}_k^{(t)})}{\sum_{i\in G}\exp(\mathbf{l}_i^{(t)}+\delta)+\sum_{i\in R}\exp(\mathbf{l}_i^{(t)})}, & k \in R. \end{cases}$$
7: Sample the next token, $s^{(t)}$, using the watermarked distribution $\hat{\mathbf{p}}^{(t)}$.
8: **end for**

III. Proposed Architecture and Algorithm Co-design

A. Efficeint Hash model for Green List Generation and Lookup

As the original lookup operation takes significant amount of memory search and value comparison, e.g. O (size of green list), for each token, to ease the major computational bottleneck shown in Fig. 2(b), a hardware efficient hash function is proposed in this work to speed up the green list generation and lookup operation. A hash function returns the lookup result in one cycle and takes an use-defined key (16 bit in the proposed scheme) for private green list generation. The main properties of a hash function include:

1) One-way property: for a mathematical function converting a variable-length input to a fixed-length result that is computationally difficult to invert.
2) Weak Partial preimage resistance: for a given hash function h(x) for input x, it is infeasible to find any partial input that can be used to construct a full input that hashes to a given hash value.
3) Weak Collision resistance: for a given input and its hash, it is difficult to find a different input b such that:

$$h(a) = h(b), a \neq b \quad (2)$$

In this work, we choose to implement an efficient encrypted hash function based on Toeplitz hash scheme which is more hardware friendly for implementation. The Toeplitz function is described as the following:

Let A be a n×n matrix. The Toeplitz matrix is defined as:

$$A = \begin{pmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{pmatrix} \quad (3)$$

The Toeplitz hash function has the property of $A_{i,j} = A_{i+1,j+1}$ and for each column in the matrix, the value is obtained from the consecutive column by shifting down 1 bit and adding one bit at the top.

Let M be the input key consisting of m bits data. Define $h_A(M)$ as the hash result by multiplying A with transposed M as shown in Algorithm 2.

To support the hash function presented in Algorithm 2, (m+n-1) bits from hardware are needed to define the whole Toeplitz matrix instead of n×m bits, which results in a significant saving in the hardware resources. Linear Feedback Shift Register (LFSR) circuit is implemented in this work for the generation of the Toeplitz matrix because of its low power and low hardware resources properties [8, 9]. The Toeplitz matrix is formed by providing the initial state of the LFSR starting from the bottom as the first column of the matrix and shifting down the previous column by one bit and adding the new bit from LFSR to the top each clock cycle.

Algorithm 2 Toeplitz Hash Function

procedure ToeplitzHash($M, h(x), s$)
  $n \leftarrow degreeof h(x)$
  $m \leftarrow lengthofmessage M$
  Initialize LFSR with $h(x)$ and state $s = (s_0, s_1, \ldots, s_{n-1})$
  Generate LFSR sequence $(s_0, s_1, \ldots, s_{m+n-2})$
  Initialize Toeplitz matrix T of size $n \times m$
  for $i = 0$ to $n - 1$ **do**
    for $j = 0$ to $m - 1$ **do**
      $T[i, j] \leftarrow s_{i+j}$
    end for
  end for
  Initialize hash value $H \leftarrow 0$
  for $j = 0$ to $m - 1$ **do**
    $H \leftarrow H \oplus (M[j] \cdot T[:][j])$
  end for
  return H
end procedure

However, by limiting the connections of the LFSRs to irreducible polynomials, the distribution of the hash function is ε-balanced for a small ε [x]:

$$\forall M \neq 0, c, P_{r_h}(h(M) = c) \leq \varepsilon \quad (4)$$

$$\varepsilon \leq \frac{m+n}{2^{n-1}} \quad (5)$$

where M is the Toeplitz matrix, $P_{r_h}(h(M))$ denotes the probability of $h(M)$ when $h \in H$, where H a set of hash functions. For a 16-bit input with a 32×32 Toeplitz hash matrix, the maximum deviation $\varepsilon \approx 0.000000022$, minimizing the probability of any particular outcome deviating significantly from the average probability of all outcomes.

B. Pruned Bitonic Sorting Network (PBSN)

The watermarking algorithm, as introduced in Section II, necessitates sorting the vocabulary based on the logit vector derived from the Transformer for checking the most possible token candidates in green list. The sorting operations are the second largest computing tasks in watermarking process due to the large size of its target list, i.e. 65536 words in the vocabulary.

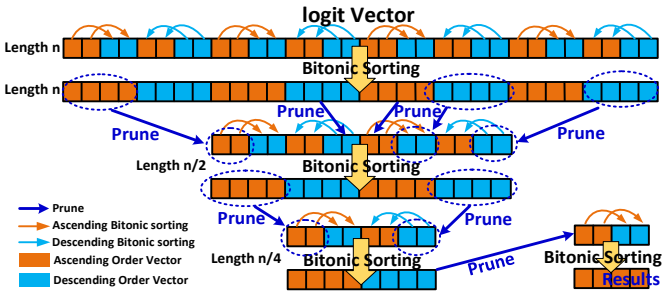


Fig. 3. Flow chart for PBSN algorithm.

The indexes corresponding to the largest k logit values can provide fixed biasing to its logit values if the indexes are shown in the green list as well. Sorting a large vocabulary logit vector and retrieving the largest k values demands a highly performant algorithm. In comparison to widely used sorting algorithms such as quicksort, merge sort, selection sort, and heapsort, the bitonic sorting network [10, 11, 12] exhibits superior parallelism suitable for specialized hardware implementation. Given that our target platform is an FPGA, which offers hardware support for parallel processing and storage, we choose bitonic sorting network as the baseline sorting method. As shown in Fig. 3, the key principle underlying the bitonic sorting network involves initially sorting a logit vector in "ascending-descending" order and subsequently utilizing the bitonic merging function to sort it into "ascending" order. For the large vocabulary logit vector, the bitonic sorting network initiates the sorting process by handling small sub-vectors with a length of 4. It then iteratively extends this sorting procedure to larger sub-vectors until the entire logit vector is effectively sorted.

Algorithm 3 Pruned Bitonic Sorting Function

Input: *array* : the 1-D array need to be sorted
  k : largest k values in the array
Output: *array* : sorted array

direction = *ascending* **or** *descending*
n = length(*array*) = 2^m, m is an integer and $m \geq 2$
BitonicSort(*array*, *ascending*, *k*)
function BitonicSort(*array*, *direction*, *k*)
  BitonicSort(*array*[1 : n/2], *ascending*, *k*)
  BitonicSort(*array*[(n/2)+1 : n], *descending*, *k*)
  PrunedBitonicMerge(*array*, *direction*, *k*)
  return *array*
end function
function PrunedBitonicMerge(*array*, *direction*, *k*)
  for $i \leftarrow 1$ **to** $n/2$ **do**
    if *array[i] and array[i+n/2] are in a wrong order* **then**
      Exchange *array* [i] and *array* [i+n/2]
  end for
  PrunedBitonicMerge(*array*[1 : n/2], *direction*)
  PrunedBitonicMerge(*array*[(n/2)+1 : n], *direction*)
  if $n \geq k$ **then**
    ascending: array = *array* [n-k : n]
    descending: array = *array* [1 : k]
  return *array*
end function

To optimize the efficiency of the watermarking, only k, e.g. 64 most possible tokens are checked against the green list. To sort and pick indexes of the largest k logit values, we devised a pruned bitonic sorting network, or PBSN. The proposed PBSN method aims to reduce processing time and eliminate redundant sorting efforts for logit values that cannot possibly be among the top k values in the final results. The details of the PBSN are introduced in Algorithm 3. Within the bitonic merging function, a length check is implemented after sorting of the sub-vector, selectively retaining only the largest k values and disregarding the remaining values in

the sub-vector. Intuitively, if an element does not belong to the largest k values within a sub-vector, it cannot be among the largest k values for the entire vocabulary logit vector. In Table 1, a comparison of average time and space complexity with other well-known sorting algorithms is presented. The proposed pruned bitonic sorting algorithm theoretically achieves significant latency improvement compared with bitonic sorting without pruning, given that k in the watermarking algorithm is notably smaller than the logit vector length n, e.g. 65536. The complexity for bitonic sorting assumes using $n/2$ comparators whose hardware cost will be described in the next section.

Table 1. Time and space complexity of pruned bitonic sorting compared with existing sorting algorithms.

Algorithm	Bubble Sort	Quick Sort	Merge Sort	Bitonic	Pruned Bitonic
Time Complexity	$O(n^2)$	O(n logn)	O(n logn)	$O(\log^2 n)$	$O(\log(n/k+1)*\log^2 k)$
Space Complexity	O(1)	O(logn)	O(n)	$O(n\log^2 n)$	$O(n\log^2 k)$

C. *Top-level Architecture for LLM-MARK*

Fig. 4 shows the top-level architecture of the proposed watermarking processor design targeting FPGA implementation. The input data is the logic vector from LLM output, which is stored in prediction buffer while the vocabulary list is stored in the watermark buffer. The Pruned Bitonic Sorter sorts the logit vector from the prediction buffer and the sorting result is sent to Hash Network for lookup. The LLM-MARK Control updates the lookup result to the watermarking list to modify the logit vector in the prediction buffer for the next iteration input to LLM.

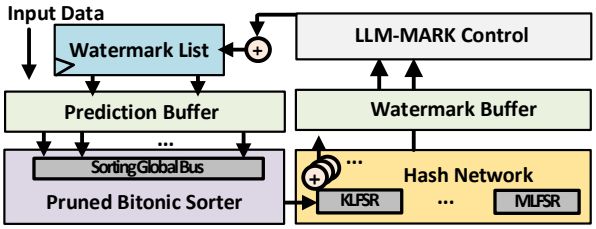


Fig. 4. Top-level Architecture of proposed watermarking processor.

IV. Hardware Mapping and Evaluation Result

In this section, the hardware mapping, quantization effects and evaluation results are shown. The design is implemented on a Xilinx XCZU15EG UltraScale+ FPGA for evaluation. Fixed integer numbers are used for watermarking process compared with floating point in baseline algorithm.

A. *Quantization Impact on Watermarking*

The logit vector from LLM model represents the probability values for each possible token which will be processed by the proposed watermarking processor. Due to the fixed integer representation in this work, the probability value is quantized leading to loss of the differentiation among the words in the green list. Fig. 5 shows the quantization loss and overall watermarking confidence with different precision and different γ (the percentage of green list size to the overall vocabulary size) during watermarking with the text input from the C4 (Colossal Clean Crawled Corpus) dataset [13].

As shown in Fig. 5(a), the cosine similarity for the green list generation during the first iteration starts to degrade from 12 bits. As shown in algorithm 1, the green list are used for logit vector probability modification, by doing so, the watermarking results are observed with deviation, as plotted in Fig. 5(b). For all testing cases, there is no watermarking loss for 16-bit compared with fp32, and there is a large similarity drop around 10-bit which will cause the watermarked text diverged drastically from the original watermarked text. The z score also starts to degrade from 10-12 bits.

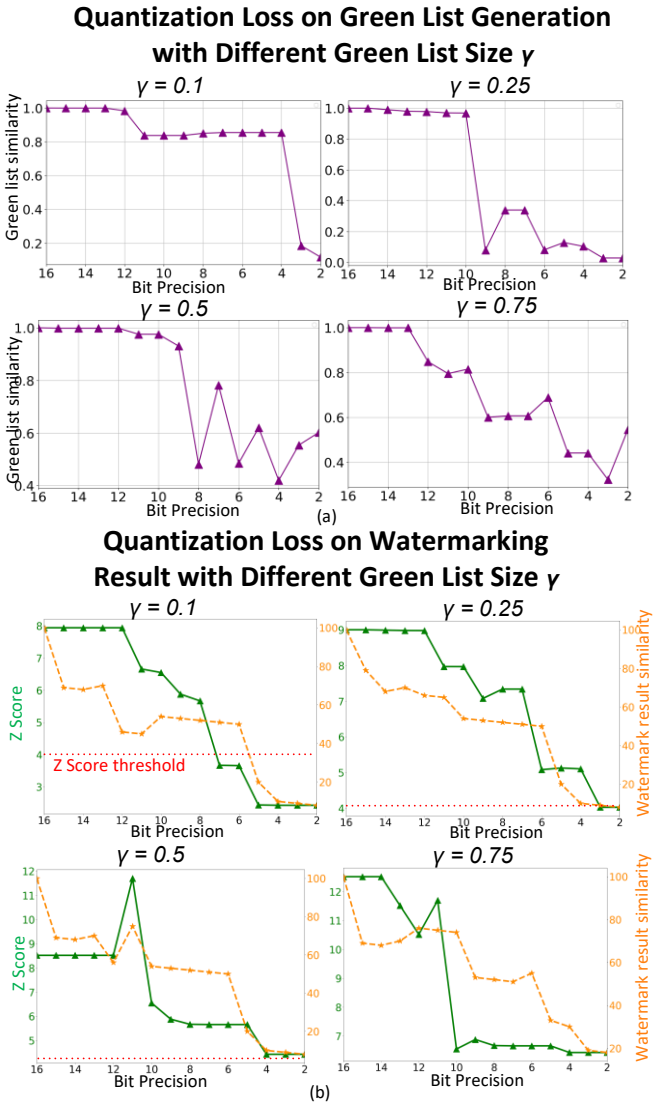


Fig. 5. (a) Quantization loss on green list generation with different green list size (first iteration). (b) Quantization loss on watermarking result with different green list size (Vocabulary size = 50272).

B. *Toeplitz Hash generation and lookup for Green list generation*

Fig. 6 shows the HW implementation of LFSR based Toeplitz hash function. The Matrix LFSR (MLFSR) is used to construct Toeplitz Hash Matrix introduced in algorithm 2. Each consecutive state in MLFSR represents each column of Toeplitz Hash Matrix with a feedback polynomial of degree 2n. The Key LFSR (KLFSR) is used to encode the input key by XOR the key with the current state and the feedback output of the KLFSR and is fed to Toeplitz Hash Matrix for the result key stream for hash lookup.

By doing so, the periodicity of the proposed hash model is $(2^{2n}-1)(2^m-1)$ where n is the column length of the Toeplitz matrix, m is the bit length of the input key.

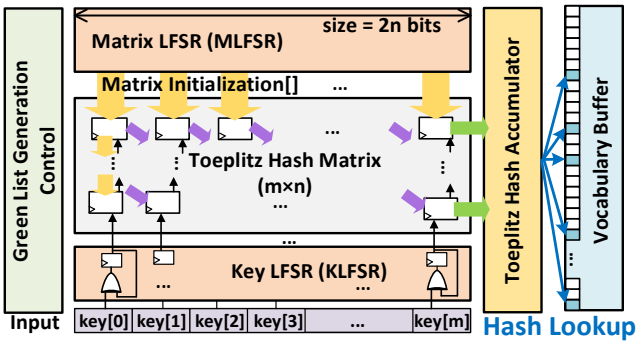


Fig. 6. Diagram for LFSR based Toeplitz hash

During green list generation, the throughput and efficiency of green list generation is defined as:

$$T_g = \frac{\text{Number of green list token generated}}{\text{Clock cycle taking to generate green list}} \quad (6)$$

$$E_g = \frac{\text{Target greenlist size}}{\text{Total hash computing time}} \quad (7)$$

For the most efficient and fast green list generation, we want to improve $T_g$with the highest E_g. Fig. 7(a) shows hash throughput with different hash matrix size compared with the non-pipelined baseline scheme. With different column + row bits numbers (26, 28, 30, 32), the average throughput of the hash function increases by 14.5X. Fig. 8(b) shows the green list generation speed up compared with the baseline scheme (Fisher-Yates shuffle [14]). The average speed up is 13.5X with γ = 0.1, 0.25, 0.5, 0.75.

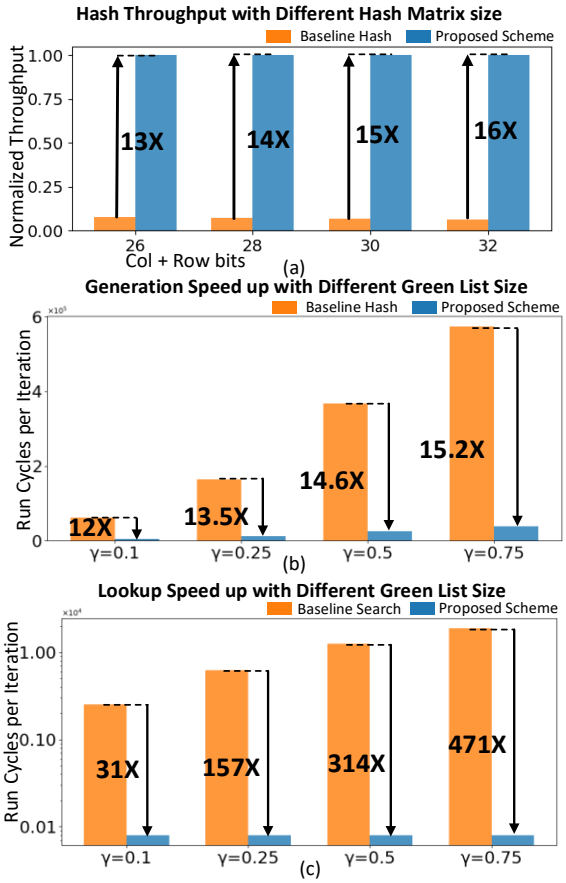


Fig. 7. (a) Hash thoughput comparison. (b) Green list generation speed up comparison. (c) lookup speedup on Xilinx XCZU15EG FPGA.

The proposed scheme is implemented on Xilinx XCZU15EG FPGA, with pipelined generation and lookup scheme. As shown in Fig. 7(c), the average speed up for different green list sizes for look up is 243X, eliminating the major overhead of execution time for green list generation and look up.

C. *Pruned Bitonic Sorter Implementation*

As illustrated in Fig. 8, the proposed sorter employing the pruned bitonic sorting algorithm is constructed using 128 block RAMs and a 128-element 16b digital comparator array in the FPGA. Each logit data has a 16b index and a 16b value. A dedicated 32*256b bus is designed to fetch the logit data from block RAMs and store the partial sum results. Given the gap between the substantial size of the vocabulary logit vector and the limited hardware resources, multiple cycles, as well as the input data management module and the finite state machine controller, are required to complete a single iteration of the bitonic sorting algorithm. The pruning function is integrated into the digital comparator to selectively retain only the top k values from the comparator output.

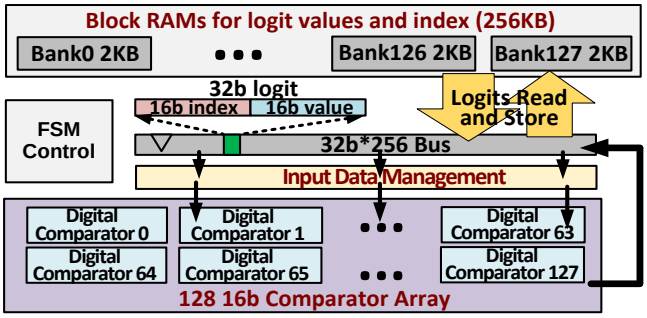
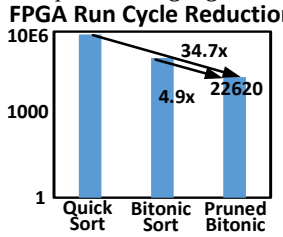


Fig. 8. Architecture diagram for pruned bitonic sorter.

In the watermarking algorithm, the k is 40 and the length of the vector (n) is 50272 based on the original algorithm [6]. One limitation of bitonic sorting is that the n must be the power of 2 and we address this through padding zeros. The FPGA evaluation results for pruned bitonic sorter are shown in Fig. 9. The pruned bitonic sorter achieves 4.9x run cycle reduction compared with bitonic sorter without pruning and 34.7x speed up over the widely used quick sorting algorithm.



FPGA Evaluation

	LUT	FF	Block RAM	I/O
Usage	60467	8928	128	67
Utilization	17.7%	1.31%	17.2%	20.4%

Fig. 9. FPGA evaluation for pruned bitonic sorter.

D. *FPGA Prototyping*

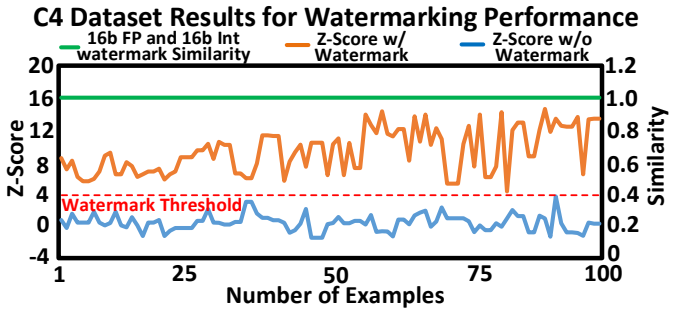
The design of the proposed watermarking processor is implemented on a Xilinx XCZU15EG FPGA board. Fig. 10 shows a board that contains one Zynq UltraScale+ 15EG chip, 2GB of DDR4 DRAM, and 3.7MB of SRAM. The watermarking design is implemented on the board with 16-bit precision to pessimistically prevent text degradation due to quantization loss, with the greenlist size of 12566, vocabulary size of 50272 and an average of 80 new tokens. 100 randomly selected results from C4 dataset are shown in Fig. 10 (b). For watermarking with 16 bits, all the generated texts are the same as the floating point watermark result. The z scores for all the watermarked text are above 4.0, and the z score for unwatermarked text are all below 4, thus the confidence for the detection is 100%. The overall run cycle using C4 dataset is shown

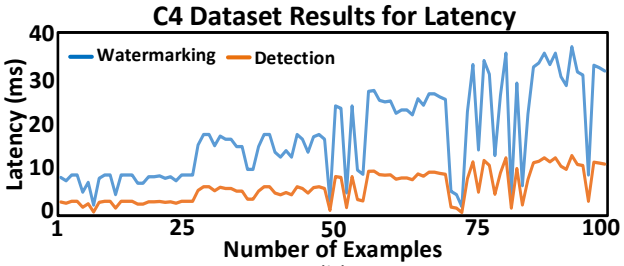
in Fig. 10(b). Including the LLM execution, each generated token has an average of 0.052s latency. The proposed scheme has 30X speed up over watermarking generation and 22.8X speed up over watermarking detection compared with the baseline implementation on the FPGA. Fig. 10(d) shows the overall hardware resources for this design.

FPGA Specification

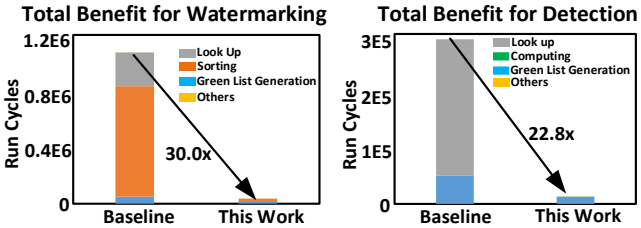
SoC	XCZU15EG-2FFVB1156I
Logic Cells	747K
DDR4	PL 2GB DDR4, 32bit
BRAM	26.2Mb

(a)





(b)



(c)

Resources	LUTs	FFs	BRAM	I/O
LLM-MARK	61063	9178	133	67
Utilization (%)	17.9	1.4	17.8	20.4
Power (W)	0.859	0.941	0.4	0.627

(d)

Fig. 10. (a) FPGA module in this work; (b) Latency and z-score of the dataset examples; (c) Overall benefits of the proposd design; (d) Resource usage on the FPGA.

V. Conclusion

In this work, we propose a hardware efficient watermarking computing architecture framework with real-time processing on local devices for the LLM authentication. By implementing a toeplitz hash generation algorithm for green list generation and look up procedure, and a pruned bitonic sorting network to sort vocabulary logit vector, this architecture framework can achieve 30x latency reduction for text watermarking and 22.8x latency reduction on watermark detection with the demonstration on Xilinx XCZU15EG FPGA. The proposed algorithm to architecture codesign framework offers a practical solution to the Intellectual Property protection in the era of generative AI.

Acknowledgement

This work was supported in part by NSF CCF-2008906.

References

[1] N. Sheybani, Z. Ghodsi, R. Kapila, and F. Koushanfar, "ZKROWNN: Zero Knowledge Right of Ownership for Neural Networks," in 2023 60th ACM/IEEE Design Automation Conference (DAC), Jul. 2023, pp. 1–6. doi: 10.1109/DAC56929.2023.10247798.

[2] S. Gehrmann, H. Strobelt, and A. Rush, "GLTR: Statistical Detection and Visualization of Generated Text," in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, M. R. Costa-jussà and E. Alfonseca, Eds., Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 111–116. doi: 10.18653/v1/P19-3019.

[3] X. Zhao, P. Ananth, L. Li, and Y.-X. Wang, "Provable Robust Watermarking for AI-Generated Text." arXiv, Oct. 13, 2023. doi: 10.48550/arXiv.2306.17439.

[4] "New AI classifier for indicating AI-written text." Available: https://openai.com/blog/%20new-ai-classifier-for-indicating-ai-written-text.

[5] E. Mitchell, Y. Lee, A. Khazatsky, C. D. Manning, and C. Finn, "DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature," in Proceedings of the 40th International Conference on Machine Learning, PMLR, Jul. 2023, pp. 24950–24962. Available: https://proceedings.mlr.press/v202/mitchell23a.html.

[6] J. Kirchenbauer, J. Geiping, Y. Wen, J. Katz, I. Miers, and T. Goldstein, "A Watermark for Large Language Models." arXiv, Jun. 06, 2023. doi: 10.48550/arXiv.2301.10226.

[7] S. Zhang et al., "OPT: Open Pre-trained Transformer Language Models." arXiv, Jun. 21, 2022. doi: 10.48550/arXiv.2205.01068.

[8] E. Erkek and T. Tuncer, "The implementation of ASG and SG Random Number Generators," in 2013 International Conference on System Science and Engineering (ICSSE), Budapest, Hungary: IEEE, Jul. 2013, pp. 363–367. doi: 10.1109/ICSSE.2013.6614692.

[9] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based Key Generator for the Niederreiter Cryptosystem Using Binary Goppa Codes," in Cryptographic Hardware and Embedded Systems – CHES 2017, vol. 10529, W. Fischer and N. Homma, Eds., in Lecture Notes in Computer Science, vol. 10529. , Cham: Springer International Publishing, 2017, pp. 253–274. doi: 10.1007/978-3-319-66787-4_13.

[10] A. Salihu, M. Hoti, and A. Hoti, "A Review of Performance and Complexity on Sorting Algorithms," in 2022 International Conference on Computing, Networking, Telecommunications & Engineering Sciences Applications (CoNTESA), Dec. 2022, pp. 45–50. doi: 10.1109/CoNTESA57046.2022.10011382.

[11] "CPP11sort: A parallel quicksort based on C++ threading - Langr - 2022 - Concurrency and Computation: Practice and Experience - Wiley Online Library." Available: https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.6606

[12] M. F. Ionescu and K. E. Schauser, "Optimizing parallel bitonic sort," in Proceedings 11th International Parallel Processing Symposium, Genva, Switzerland: IEEE Comput. Soc. Press, 1997, pp. 303–309. doi: 10.1109/IPPS.1997.580914.

[13] "allenai/c4 · Datasets at Hugging Face." Available: https://huggingface.co/datasets/allenai/c4

[14] R. Durstenfeld, "Algorithm 235: Random permutation," Commun. ACM, vol. 7, no. 7, p. 420, Jul. 1964, doi: 10.1145/364520.364540