Deep Learning-Driven FPGA Function Block Detection Using Bit stream-to-Image Transformation

M.Shiva Priya
Assistant professor of CSE
Malla Reddy College of Engineering,Maisammaguda, Hyderabad.
PhD scholar of NirwanUniversity,Jaipur

Dr Venkata Shesha Giridhar Akula
Principal
Sphoorthy Engineering College Nadergul, Hyderabad

Dr. P. Sammulal
Professor of CSE
JNTUH ,University College of Engineering, Science & Technology, Hyderabad

ABSTRACT

To enhance the security of Field Programmable Gate Arrays (FPGAs) across various application scenarios, it is essential to analyse the system functions of FPGA designs. This can be achieved by partitioning the FPGA bitstream into functional blocks and identifying their functionalities. In this study, we present a novel deep learning-based method for FPGA function block detection, which involves three key steps.First, we analyze the bitstream format to establish the mapping between configuration bits and configurable logic blocks, addressing the discontinuity of configuration bits for individual elements. Next, we leverage advancements in deep learning-based object detection by transforming the FPGA bitstream into an image. This transformation method considers both the adjacency of programmable logic and the redundancy in configuration information. Once the bitstream is converted into an image, a deep learning-based object detection algorithm is applied, enabling the identification of function blocks within the original FPGA design.The deep neural network employed for detection is trained and validated using a custom bitstream-to-image dataset. Experimental results demonstrate the effectiveness of this method, achieving a mean Average Precision (mAP) of 98.11% (IoU = 0.5) for detecting 10 function blocks using a YOLOv3 detector implemented on Xilinx Zynq-7000 SoCs and ZynqUltraScale+ MPSoCs.

Keywords:Bitstream-to-image transformation, field-programmable gate array, function block detection.

I. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are increasingly utilized in diverse fields such as communication, deep learning, and digital signal processing, thanks to their unique advantages of configurability, rapid development cycles, and abundant logic and storage resources. However, with the growing adoption of FPGA technologies comes a significant security challenge for FPGA-based systems.

The behavior and functionality of an FPGA design are defined entirely by its bitstream, which contains the configuration data loaded during power-on. This makes the bitstream a critical vulnerability in FPGA security. Assessing the security of an FPGA design often involves reverse-engineering the bitstream to identify and isolate functional blocks and determine their intended operations.

A function block in an FPGA design refers to a circuit block implementing a specific function, such as a cryptographic operator like MD5 (Message Digest Algorithm 5). FPGA applications typically include various types of function blocks. For instance, an FPGA implementation of the PDF-R2 encryption algorithm contains two key function blocks: MD5 and RC4 (Rivest Cipher 4). Detecting these function blocks is an essential pre-analysis step in understanding the system's overall functionality.

One common method for identifying function blocks involves partitioning the circuit, represented by bitstreams or netlists, and comparing the partitioned sections against known designs. However, this approach faces challenges due to the time-consuming partitioning process, which often produces imperfect results and incorrect matches. Furthermore, traditional methods require manually designing comparison features, and poorly selected features can degrade performance.

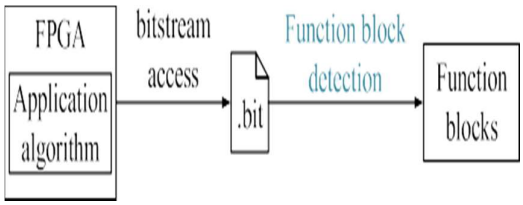

Fig. 1 illustrates an example application scenario where FPGA function block detection is utilized to analyze the system functionality of a circuit.

To address these limitations, we propose a novel FPGA function block detection method that identifies one or more function blocks directly from a complete bitstream. By transforming the bitstream into an image, we recast the FPGA function block detection problem as an image object detection task, effectively leveraging deep learning techniques.

This transformation ensures that the adjacency of programmable logic elements is preserved in the resulting image. Two critical challenges complicate this process: the high redundancy of configuration data and the discontinuity of configuration bits for individual elements. To overcome these challenges, our approach focuses exclusively on the configuration bits of Configurable Logic Blocks (CLBs) and maps these bits to their corresponding CLBs, enabling a more accurate and efficient transformation.

In summary, the main contributions of this paper are as follows:

(a) Bitstream-to-ImageTransformation:

A novel method for transforming bitstreams into images suitable for deep learning is proposed. This is achieved by analyzing the mapping relationship between configuration bits and CLB elements.

(b) Dataset Generation: A dataset is created by converting bitstreamfiles into images representing 10 types of cryptographic operators. This approach eliminates the need for manual annotation, as the data is automatically labeled.

(c) Deep Learning for FPGA Function Block Detection: For the first time, deep learning techniques are applied to detect FPGA function blocks directly from bitstreams. A deep learning-based object detection algorithm is trained on the generated dataset. Experimental validation using various FPGA designs for application-specific encryption algorithms demonstrates the effectiveness of the method, achieving a mean Average Precision (mAP) of 98.11% for 10 types of function blocks at an Intersection over Union (IoU) threshold of 0.5.

The rest of the paper is structured as follows: Section II outlines the overall detection process and provides a detailed explanation of each step. Section III presents the experimental results, while Section IV concludes with a summary of findings.

II. PROPOSED FUNCTION DETECTION METHOD THROUGH BITSTREAM TO IMAGE TRANSFORMATION

The process of function block detection involves the following steps, as illustrated in Fig. 2:

- **Bitstream-to-Image Transformation**: The process begins by analyzing the mapping relationship between configuration bits and CLBs (Configurable Logic Blocks) to transform the bitstream into an image.
- **Deep Learning Inference**: The generated image is then fed into a trained deep learning model to identify the function blocks within the bitstream.

- **Model Training**: The deep learning model is trained using a dataset composed of numerous images derived from various bitstreams.

A. TRANSFORMATION FROM BITSTREAM TO IMAGE

When transforming a bitstream file into an image, two challenges need to be addressed.

The first challenge is that the configuration bits for a single element are not stored consecutively in the bitstream. The configuration memory in the bitstream is organized into frames, the smallest addressable segments of memory. For example, the configuration bits for one Configurable Logic Block (CLB) are distributed across multiple frames. To extract features from the transformed image, it must reflect the adjacency of elements in the programmable logic, representing the physical proximity of the utilized elements. This adjacency is influenced by optimization algorithms used by EDA tools during placement and routing. While implementations from different EDA tools or repeated runs of the same function block share many common features, they are not identical. The discontinuous nature of configuration bits complicates representing the adjacency between elements in the programmable logic.

The second challenge lies in the fact that not all configuration data in the bitstream is relevant to function block detection, resulting in two negative effects:

Since this work focuses on logic resources, data from other resources can introduce confusion. The programmable logic of an FPGA includes CLBs, Input/Output Blocks (IOBs), Block RAMs (BRAMs) for dense storage, and Digital Signal Processors (DSPs) for high-speed computing. For example, BRAM utilization depends on the array size of function blocks, so function blocks of the same type may have varying BRAM usage. However, the logic remains unaffected by data size. Configuration bits unrelated to logic resources that do not aid function block detection must be identified and excluded.

The large size of transformed images creates a dataset that is difficult to load efficiently during deep learning training. For instance, converting the entire bitstream of a Xilinx Zynq-7000 SoC ZC702 Evaluation Board into a three-channel color image produces a 1280 × 1080 × 3 image, which is too large for effective processing by the detection model.

These challenges are addressed with a proposed bitstream-to-image transformation. First, the bitstream format is analyzed to map the relationship between CLB elements and their configuration bits. Second, only the configuration bits of CLBs are used for representation, effectively compressing the data and discarding irrelevant information.

1) MAPPING RELATIONSHIP BETWEEN CONFIGURATION BITS AND CLBs

An FPGA bitstream consists of three main parts: the Head-of-File, FDRI (Frame Data Register Input) data, and End-of-File. Among these, the FDRI data contains the configuration information for the programmable logic.

The programmable logic is divided into multiple Clock Regions, each containing several columns of Configurable Logic Blocks (CLBs). As illustrated in Fig. 3, a column of CLBs consists of qqq CLBs. Each frame within the bitstream contains mmm 32-bit words.

Although the open official documentation does not explicitly define how CLBs are configured with FDRI data, it has been observed that every successive nframes of FDRI data configure a single column of CLBs (q CLBs). Apart from l words located in the middle of the frame, every p words in the remaining m−l words of a frame correspond to a single CLB, organized from the bottom to the top of the column.

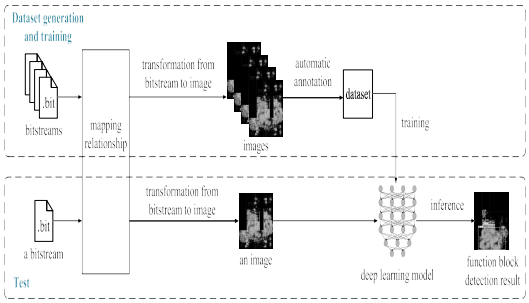

Fig. 2. The process of function block detection method consisting of the transformation from bitstream to image, dataset generation, deep learning training, and deep learning inference.

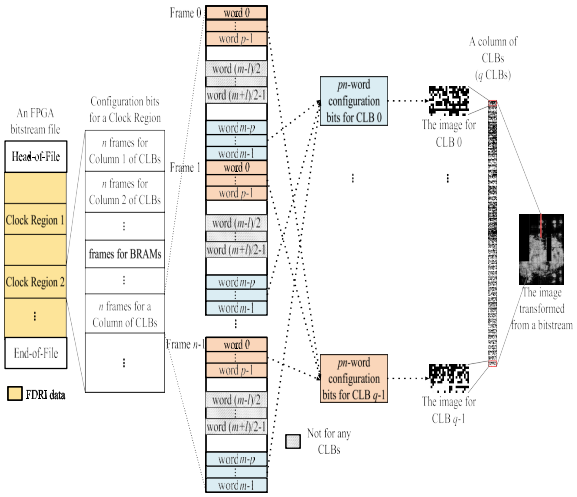

Fig. 3 The transformation from an FPGA bitstreamfile to the image.

For two-slice CLBs, the ppp words configure the two slices separately, from left to right. Consequently, each CLB in a column requires p×nwords to configure, with these words distributed at the same positions across the n frames.

2) TRANSFORMATION FROM CONFIGURATION BITS FOR CLBs TO IMAGES FOR CLBs

Based on the analysis of the bitstream format, each CLB is allocated 4p×nbytes of configuration memory from successive nnn frames. For two-slice CLBs, each slice is allocated s=2p×n bytes. In a one-slice CLB, each slice is allocated s=4p×n bytes. Since the

configuration bits are used to configure individual slices, the bits with non-zero values indicate that the respective slice is utilized in the implementation.

There are three methods to transform the s-byte configuration data of a slice into a separate image with the appropriate height and width, which can represent the utilization of the slice.

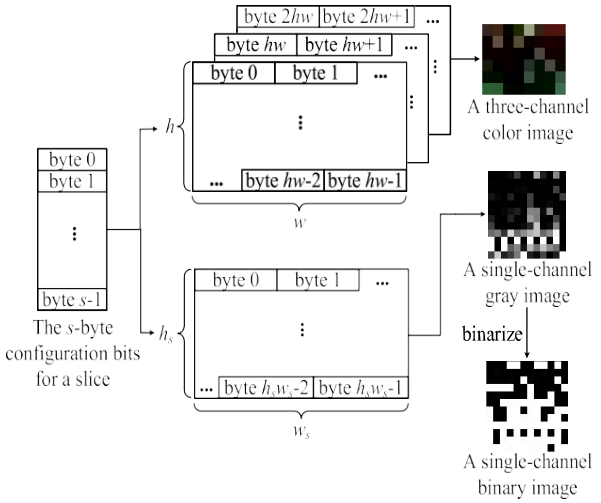

Fig 4. The s-byte configuration bits for a slice are transformed into a three-channel color image with h×w pixels, a single-channel gray image with hs×ws pixels or a single-channel binary image with hs×ws pixels.

The first method involves transforming the data into a three-channel color image. As shown in Fig. 4, the configuration bits of a slice can be converted into a three-channel color image with dimensions h×w pixels. Each pixel, having three channels, requires three bytes. The configuration bits are treated as image data arranged in a three-dimensional format of (channel, height, width).

The second method converts the data into a single-channel grayscale image. Unlike the three-channel color image, the data for each pixel in the grayscale image corresponds directly to continuous configuration data for a slice. The configuration bits of a slice are transformed into a grayscale image with dimensions $h_s \times w_s$ pixels, where each pixel requires one byte of storage.

The third method transforms the data into a single-channel binary image. This binary image is derived from the binarization of the grayscale image. Pixels with non-zero values in the grayscale image are set to one in the binary image, while others are set to zero. The binary image uses only zeros and ones to indicate whether the slices are utilized.

3) STRUCTURE OF THE IMAGE TRANSFORMED FROM BITSTREAM

The image transformed from an FPGA bitstream is divided into a×bblocks, with each block corresponding to a CLB in one of the Clock Regions. The parameters a andb are device-specific and remain consistent across all three methods of transforming configuration bits into images for CLBs.

As illustrated in Fig. 3, the transformation process is performed individually for each CLB. The resulting images for all CLBs are

then combined to construct the complete image. The adjacency between CLBs in the programmable logic is reflected in the aggregation of individual CLB images into the overall image.

B. DEEP LEARNING MODEL AND TRAINING

Since deep learning techniques have not previously been applied to FPGA function block detection, our approach leverages the transformation of bitstreams into images along with the image feature extraction capabilities of deep neural networks (DNNs). YOLOv3 [20] and SSD [21] are two classical one-stage object detection algorithms based on deep learning, offering a combination of high speed and accuracy.

1) YOLOv3

YOLOv3 consists of 75 convolutional layers, with its backbone architecture based on Darknet-53 [20]. It has three output convolutional layers, each producing feature maps of different sizes to detect objects of varying scales. These output layers have the same number of filters, determined by the number of classes.

The input image is divided into grids, with each grid cell predicting box_number bounding boxes (in YOLOv3, box_number is 3). For each bounding box, the model predicts one objectness score, CCC class probabilities for CCC classes, and 4 box offsets. The objectness score quantifies the likelihood that the bounding box contains a generic object [22]. Consequently, the filter number for the output layers is $3\times(1+C+4)$. For instance, when applying YOLOv3 to detect 10 types of function blocks ($C=10C = 10C=10$), the filter number of the output layers is 45. The input channel number for the first convolutional layer is determined by the number of channels in the images fed into the neural network.

During training, YOLOv3 uses pre-trained weights from the COCO dataset [23] as initial weights. In the first 50 epochs, the earlier layers are frozen except for the last three output convolutional layers to stabilize the loss value. From the 51st to the 100th epoch, all layers are unfrozen and trained with a reduced learning rate. After training, the model with the smallest validation loss is selected as the final model.

2) SSD

SSD consists of 29 convolutional layers and 4 max-pooling layers, with its backbone architecture based on VGG-16 [24]. It includes six output convolutional layers designed to detect objects of different sizes.

Similar to YOLOv3, the filter number in SSD's output layers is calculated asbox_number×(C+4). In SSD, box_number can be either 4 or 6, depending on the output layer. For example, when C=10C = 10C=10, the filter number for the output layers is 56 or 84.

During training, SSD uses the pre-trained weights of VGG-16 for the initial weights of the front layers. In the first training stage, these front layers are frozen to retain the VGG-16 weights. In the second training stage, the entire network is unfrozen and trained as a whole.

3) GENERATION OF DATASET

Bitstreams are transformed into images, which are compiled into a dataset for deep learning training and testing. Each bitstream file implements an algorithm, and each algorithm contains one or more function blocks. In practical applications, a single type of function block may have different variations, such as a standard design and a pipelined version. Consequently, each type of function block is implemented in one or two variations during bitstream generation.

To train the deep network, multiple bitstream files containing various types of function blocks are required. A large number of bitstreams are generated using the Xilinx Vivado EDA toolset. Constraints on the implementation region ensure that function blocks are placed in different locations across different bitstreams. Instead of relying on the graphical user interface (GUI), Tcl (Tool Command Language) [25] scripts are utilized to automate the process. These scripts also extract the categories and locations of function blocks in the FPGA device diagram during bitstream generation.

Finally, the bitstream files are converted into images using Python scripts. These scripts simultaneously process the label information and generate annotation files required for deep learning.

IV. EXPERIMENTAL RESULTS

A. EXPERIMENTAL SETUP

For evaluation, we utilize Xilinx Zynq-7000 SoCs and Xilinx ZynqUltraScale+ MPSoCs to test our proposed methodology. Unless explicitly stated otherwise, all experiments in this section are conducted under the following setup:

Bitstream files are generated without encryption using the Xilinx Vivado design suite, including versions Vivado 2016.3, Vivado 2017.2, Vivado 2017.4, Vivado 2018.3, and Vivado 2019.2.Scripts for transforming bitstreams into images are executed in Python 2.7.15.The training and testing of the deep learning models are

performed using Keras 2.2.5, based on TensorFlow 1.10.0 for GPUs, with Python 3.5.6.A server running CentOS Linux 7.6, equipped with an NVIDIA Tesla P100 GPU, is used to perform all experiments.

Our methodology targets the detection of 10 types of function blocks. YOLOv3 is primarily used as the deep learning-based object detection algorithm unless stated otherwise. SSD is only used in specific experiments described in Section IV-E5. The training parameters for YOLOv3 and SSD are listed in TABLE 1.

TABLE 1.Parameters for the Training Process of YOLOv3 and SSD

Deep neural network		YOLOv3	SSD
Input size[a]		416×416×1 or 416×416×3	300×300×1 or 300×300×3
Optimizer		Adam [2o]	Adam
The first stage of training	Batch size	32	32
	Learning rate	0.001	0.001
The second stage of training	Batch size	16	32
	Learning rate	0.0001	0.001

To quantitatively evaluate the object detector's performance, we use the mean Average Precision (mAP) metric under specific Intersection over Union (IoU) thresholds, which accounts for both precision and recall. A good performance is indicated when the IoU between the detected box and the ground truth exceeds 0.5. In this work, we use two metrics inspired by the COCO dataset:

mAP@0.5, the standard metric for the PASCAL VOC dataset [27].

mAP@0.75, a stricter metric than mAP@0.5.

B. BITSTREAM FORMAT ANALYSIS

To establish the mapping relationship for transforming bitstreams into images, this work analyzes the bitstream formats of Xilinx Zynq-7000 SoCs and Xilinx ZynqUltraScale+ MPSoCs. In the case of Xilinx Zynq-7000 SoCs, each CLB element comprises two slices, with each slice consisting of 4 LUTs and 8 Flip-Flops (FFs) [28]. For Xilinx ZynqUltraScale+ MPSoCs, a CLB element contains one slice, which consists of 8 LUTs and 16 FFs [29]. These slices are of two types: SLICEL (logic) and SLICEM (memory). Detailed information regarding the bitstream format can be found in the official documentation [30], [31].

When transforming bitstreams into images, it is essential to determine the positions of the first frames, which can be identified by configuring different columns of CLBs. For example, the positions of some first frames in the ZC702 FPGA bitstream are shown in Fig. 5. For the ZC702 FPGA, the number of frames required to configure a column of CLBs (n) is 36, while 28 frames are needed to configure a column of BRAMs.

C. TRANSFORMATION RESULT FROM BITSTREAM

Based on the bitstream format analysis presented in Section I II-B, the bitstreams are transformed into images. Specifically, the analysis applies to the Xilinx Zynq-7000 SoC Z-7020, Xilinx Zynq-7000 SoC Z-7030, and Xilinx ZynqUltraScale+ MPSoC ZU9EG.

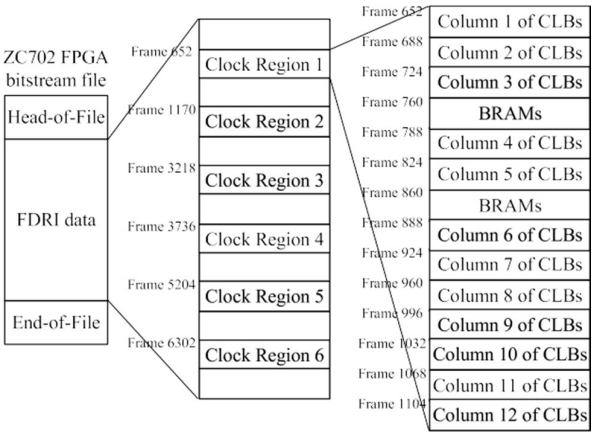

Fig.5The positions of some first frames in the bitstream of ZC702 FPGA are shown. The number of frames configuring a column of CLBs (n) is 36 for ZC702 FPGA.

TABLE 2.Parameters for Transforming the Bitstreams into Images, taking Xilinx Zynq-7000 SoC Z-7020, Xilinx Zynq-7000 SoC Z-7030, and Xilinx ZynqUltraScale+ MPSoC ZU9EG as Examples.

Device name	SoC Z-7020[a]	SoC Z-7030[b]	ZU9EG[c]
divided ($a \times b$)	150×57	200×60	51 for SLICEM)
length (bits)	32,364,512	47,839,328	212,086,240
of a slice	6×8×3	6×8×3	for SLICEM
image (bytes)	900×912×3	1200×960×3	2940×1587×3
image(%)	60.87	57.79	52.80
image of a slice	12×12×1	12×12×1	for SLICEM
(bytes)	1800×1368×1	2400×1440×1	5040×2781×1
image(%)	60.87	57.79	52.87

as examples, the parameters for transforming the bitstreams into images are listed in TABLE 2.

The bitstream length of the ZC702 FPGA is 3.86 MiB. The three-channel color image transformed from the bitstream has a size of 900 × 912 × 3 bytes. This transformation effectively compresses the color image to (900 × 912 × 3 × 8 bits / 32,364,512 bits) × 100% = 60.87% of the original bitstream data. The effective compression is primarily due to the exclusion of configuration bits that are irrelevant to the logic resources. The parameters for transforming the bitstreams into images are set differently for various FPGA devices, based on the bitstream format information specific to each device.

The Vivado-implemented design and the image transformed from the bitstream of the Xilinx Zynq-7000 SoC Z-7020 are shown in Fig. 6. The dark blue area in the Vivado-implemented design represents columns of CLBs, the configuration bits of which are used for the bitstream representation. The

corresponding relationship between the Vivado-implemented design and the image transformed from the bitstream for the same FPGA device demonstrates that the mapping relationship between the configuration bits and CLBs identified in this work is correct, and that the image transformed from the bitstream accurately reflects the adjacency of the programmable logic.

In conclusion, the two challenges discussed in Section III-A have been successfully addressed by the proposed bitstream-to-image transformation.

D. DATASET DESCRIPTION

To train and test the DNN models, a large number of bitstreams, implementing FPGA designs on the ZC702 FPGA, are generated to create the dataset. The dataset includes 15 types of application-specific encryption algorithms, which collectively contain 10 different cryptographic operators. These encryption algorithms and their associated cryptographic operators are listed in TABLE 3. Each encryption algorithm used in this work incorporates up to 3 different cryptographic operators. Each operator is implemented in one or two different constructions. "Pipelined" refers to a cryptographic operator implemented in a pipeline design, while "module" refers to an implementation without special designs. For example, a bitstream implementing the NTLM (NT LAN Manager) encryption algorithm includes either a Message Digest Algorithm 4 (MD4) pipeline or an MD4 module. Similarly, a bitstream implementing the encryption algorithm for PDF-R2 contains an MD5 pipeline and an RC4 module.

TABLE 3.Components of the Training Set and Test Set

Vivado version	alg	contain	oper	Implementation	bitstream
	For train				
2017.4	LINU	1	DES	DES pipeline; DES module	728
		1	MD4	MD4 pipeline; MD4 module	
		2	MD5, RC4	MD5 pipeline, RC4 module	
		2	MD5, RC4	MD5 pipeline, RC4 module	
		3	AES, Serpent, Twofish	AES module, Serpent module, Twofish module	
		1	SHA-1	SHA-1 pipeline	
		1	SHA-1	SHA-1 pipeline	
		1	SHA-1	SHA-1 pipeline	
		1	SHA-1	SHA-1 module	
		1	SHA-256	SHA-256 pipeline	
		1	SHA-256	SHA-256 pipeline	
		1	SHA-512	SHA-512 module	
		1	SHA-512	SHA-512 module	
		1			
	For te				
2016.3	P	2	MD5,	MD5 pipeline,	
		2			
2017.2	P	2	MD5,	MD5 pipeline,	
		2			
2017.4	OFF	2	MD5, RC4	SHA-1 p	
		1	SHA-1		
2018.3	P	2	MD5,	MD5 pipeline,	
		2			
2019.2	P	2	MD5,	MD5 pipeline,	
		2		MD5 pipeline,	
					8,239

To ensure reasonable experimental setup, 13 of the encryption algorithms listed in TABLE 3 are selected to form the training and test sets, comprising 10,047 bitstreams generated by Xilinx Vivado 2017.4. These bitstreams are randomly divided into a 4:1 ratio for the training and test sets. Two encryption algorithms used for PDF-R5 and OFFICE 2010, also implemented by Xilinx Vivado 2017.4, are exclusively used for testing.

Additionally, to investigate the impact of EDA tools, bitstreams generated by various versions of Xilinx Vivado (2016.3, 2017.2, 2018.3, and 2019.2) are used to evaluate the performance of the trained model. These bitstreams are transformed into images, which make up the dataset for training and testing.

VII. CONCLUSION

In this paper, we propose an FPGA bitstream function block detection method leveraging deep learning techniques. First, the bitstream format of FPGA designs was analyzed to determine the mapping relationship between configuration bits and CLB elements. The bitstreams of various FPGA designs were then transformed into images suitable for deep learning processing, preserving the adjacency of the programmable logic. These transformed images formed a comprehensive dataset.

In our experiment, we created a dataset comprising 18,268 images derived from multiple bitstreams. This dataset was used to train a deep learning-based object detection algorithm. Once trained, the algorithm was applied to detect function blocks directly from FPGA bitstreams. The entire process, including dataset generation, model training, and testing, can be fully automated.

Experimental results demonstrate that the proposed method achieves a mean Average Precision (mAP) of 98.11% (IoU = 0.5) across 10 types of function blocks using YOLOv3 as the object detector. Furthermore, the detector successfully identified function blocks in bitstreams for system designs not included in the training set and those generated by different EDA tools.

REFERENCES

[1] Internet Engineering Task Force. (1992).RFC 1321-The MD5 MessageDigest Algorithm.[Online]. Available: https://tools.ietf.org/html/rfc1321

[2] Cypherpunks (Mailing List). (1994). Thank you Bob Anderson.[Online]. Available: http://cypherpunks.venona.com/date/1994/09/msg00304.html

[3] D. Ziener, S. Assmus, and J. Teich, ‘‘Identifying FPGA IP-cores based on lookup table content analysis,’’ in Proc. Int. Conf. Field Program. Log. Appl., Aug. 2006, pp. 1–6.

[4] J. Couch, E. Reilly, M. Schuyler, and B. Barrett, ‘‘Functional block identification in circuit design recovery,’’ in Proc. IEEE Int. Symp.Hardw.Oriented Secur. Trust (HOST), May 2016, pp. 75–78.

[5] P. Liu, S. Li, and Q. Ding, ‘‘An energy-efficient accelerator based on hybrid CPU-FPGA devices for password recovery,’’ IEEE

Trans. Comput., vol. 68, no. 2, pp. 170–181, Feb. 2019.

[6] R. Le Roux, G. Van Schoor, and P. Van Vuuren, ‘‘Parsing and analysis of a xilinx FPGA bitstream for generating new hardware by direct bit manipulation in real-time,’’ South Afr. Comput. J., vol. 31, no. 1, pp. 80–102, Jul. 2019.

[7] K. Dang Pham, E. Horta, and D. Koch, ‘‘BITMAN: A tool and API for FPGA bitstream manipulations,’’ in Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE), Mar. 2017, pp. 894–897.

[8] L. Bozzoli and L. Sterpone, ‘‘COMET: A configuration memory tool to analyze, visualize and manipulate FPGAs bitstream,’’ in Proc. Int. Conf. Archit. Comput. Syst. (ARCS) Workshop, Apr. 2018, pp. 1–4.

[9] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, ‘‘On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs,’’ in Proc. 18th ACM Conf. Comput. Commun.Secur.(CCS), Oct. 2011, pp. 111–124.

[10] S. Tajik, H. Lohrke, J.-P. Seifert, and C. Boit, ‘‘On the power of optical contactless probing: Attacking bitstream encryption of FPGAs,’’ in Proc. ACM SIGSAC Conf. Comput. Commun.Secur., Oct. 2017, pp. 1661–1674.

[11] M. Ender, A. Moradi, and C. Paar, ‘‘Theunpatchable silicon: A full break of the bitstream encryption of Xilinx 7-series FPGAs,’’ in Proc. USENIX Secur. Symp., Aug. 2020, pp. 1803–1819.

[12] J.-B. Note and É. Rannaud, ‘‘From the bitstream to the netlist,’’ in Proc. 16th Int. ACM/SIGDA Symp. Field Program. Gate Arrays (FPGA), 2008, p. 264.

[13] F. Benz, A. Seffrin, and S. A. Huss, ‘‘Bil: A tool-chain for bitstream reverse-engineering,’’ in Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL), Aug. 2012, pp. 735–738.

[14] Z. Ding, Q. Wu, Y. Zhang, and L. Zhu, ‘‘Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation,’’ Microprocessors Microsyst., vol. 37, no. 3, pp. 299–312, 2013.

[15] T. Zhang, J. Wang, S. Guo, and Z. Chen, ‘‘A comprehensive FPGA reverse engineering tool-chain: From bitstream to RTL code,’’ IEEE Access, vol. 7, pp. 38379–38389, 2019.

[16] Y.-Y. Dai and R. K. Braytont, ‘‘Circuit recognition with deep learning,’’ in Proc. IEEE Int. Symp.Hardw.Oriented Secur. Trust (HOST), May 2017, p. 162.

[17] A. Fayyazi, S. Shababi, P. Nuzzo, S. Nazarian, and M. Pedram, ‘‘Deep learning-based circuit recognition using sparse mapping and leveldependent decaying sum circuit representations,’’ in Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE), Mar. 2019, pp. 638–641.

[18] S. Mahmood, J. Rettkowski, A. Shallufa, M. Hubner, and D. Gohringer, ‘‘IP core identification in FPGA configuration files using machine learning techniques,’’ in Proc. IEEE 9th Int. Conf. Consum.Electron. (ICCEBerlin), Sep. 2019, pp. 103–108.

[19] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E.Gaillardon, ‘‘LSOracle: A logic synthesis framework driven by artificial intelligence: Invited paper,’’ in Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD), Nov. 2019, pp. 1–6.

[20] J. Redmon and A. Farhadi, ‘‘YOLOv3: An incremental improvement,’’ 2018, arXiv:1804.02767. [Online]. Available: http://arxiv.org/abs/ 1804.02767

[21] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, ‘‘SSD: Single shot multibox detector,’’ in Proc. Eur. Conf. Comput. Vis., Amsterdam, The Netherlands, Oct. 2016, pp. 21–37.

[22] B. Alexe, T. Deselaers, and V. Ferrari, ‘‘Measuring the objectness of image windows,’’ IEEE Trans. Pattern Anal. Mach. Intell., vol. 34, no. 11, pp. 2189–2202, Nov. 2012.

[23] T. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, Dollár, and C. L. Zitnick, ‘‘Microsoft COCO: Common objects in context,’’ in Proc. 13th Eur. Conf. Comput. Vis. (ECCV), Zürich, Switzerland, Sep. 2014, pp. 740–755.

[24] K. Simonyan and A. Zisserman, ‘‘Very deep convolutional networks for large-scale image recognition,’’ in Proc. Int. Conf. Learn. Represent. (ICLR), May 2015, pp. 1–14.

[25] Xilinx. (2018). Vivado Design Suite TCL Command Reference Guide (UG835). [Online]. Available: https://www.xilinx.com/support/ documentation/sw_manuals/xilinx2017_4/ug% 835-vivado-tclcommands.pdf

[26] D. P. Kingma and J. Ba, ‘‘Adam: A method for stochastic optimization,’’ in Proc. Int. Conf. Learn. Represent. (ICLR), May 2015, pp. 1–15.

[27] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, ‘‘The PASCAL visual object classes (VOC) challenge,’’ Int. J. Comput. Vis., vol. 88, no. 2, pp. 303–338, Jun. 2010.

[28] Xilinx. (2018). Zynq-7000 SoC Technical Reference Manual (UG585).[Online]. Available: https://www.xilinx.com/support/documentation/user_ guides/ug585-Zynq-700%0-TRM.pdf

[29] Xilinx. (2017). UltraScale Architecture Configurable Logic Block User Guide (UG574). [Online]. Available: https://www.xilinx.com/support/ documentation/user_guides/ug574-ultrasca%le-clb.pdf

[30] Xilinx. (2018). 7 Series FPGAs Configuration User Guide (UG470). [Online]. Available: https://www.xilinx.com/support/documentation/ user_guides/ug470_7Series_%Config.pdf

[31] Xilinx. (2020). UltraScale Architecture Configuration User Guide (UG570). [Online]. Available: https://www.xilinx.com/support/ documentation/user_guides/ug570-ultrasca%le-configuration.pdf