JSCDM

JOURNAL OF SOFT COMPUTING AND DATA MINING
e-ISSN: 2716-621X

Vol. 7 No. 1 (2026) 37-54
https://publisher.uthm.edu.my/ojs/index.php/jscdm

Enhancing Digital Evidence Analysis: Fakeseeker for Real-Time Deepfake Detector using EfficientNet

Chua Kai Zen[1], Nurul Hidayah Ab Rahman[1]*, Niken Dwi Wahyu Cahyani[2], Mubarak Saif Mohsen[3]

[1] *Centre for CyberSecurity and Data Intelligence, Universiti Tun Hussein Onn Malaysia, Parit Raja, Batu Pahat, 86400, MALAYSIA*
[2] *School of Computing, Telkom University, Bandung, INDONESIA*
[3] *Sana's Community College, Sana'a, YEMEN*

*Corresponding Author: hidayahar@uthm.edu.my
DOI: https://doi.org/10.30880/jscdm.2026.07.01.004

Article Info

Received: 4 September 2025
Accepted: 8 December 2025
Available online: 10 April 2026

Keywords

Artificial intelligence, deep learning, forensic analysis, media analysis

Abstract

Deepfake technology poses significant challenges to the authenticity of digital evidence. This creates an urgent need for reliable and effective verification methods. This work introduces Fakeseeker, a lightweight, real-time detection framework designed to assist digital forensic investigations. The system uses a fine-tuned EfficientNet-B0 architecture to analyze both uploaded and live-streamed media. The approach involves preparing datasets from publicly available sources, including Celeb-DF, FaceForensics++, and DFDC. Subsequently, MTCNN-based face extraction, data augmentation, and targeted optimization of EfficientNet-B0 were undertaken. Comparative evaluations with EfficientNet variants B0, B1, and B2 show that EfficientNet-B0 offers the best balance between accuracy and efficiency. It takes about nine minutes per training epoch, achieves a validation LogLoss of 0.0021, and has an AUC score of 0.9996. Based on these results, Fakeseeker was built using EfficientNet-B0 as its backbone. It includes separate components for media processing, classification, and reporting. Evaluation results show a detection accuracy of 99.2% with minimal resource usage. This makes the system suitable for real-time forensic workflows and field applications. Overall, this study provides a practical and scalable solution for improving the verification and analysis of digital evidence.

1. Introduction

Deepfake technology has recently emerged as a significant digital trend [1]. Research on deepfakes in artificial intelligence began by using neural networks to create digital changes [2]. It uses deep-learning methods to manipulate video and image content. Deepfake technology refers to media produced with deep learning techniques that change or replace faces in images or videos [2]. While these technologies offer creative possibilities in entertainment and education, their ability to generate hyper-realistic but fake media poses serious risks for misuse. Additionally, the spread of deepfakes raises important social issues, such as undermining trust in digital media, spreading false information, and enabling fraud and privacy breaches.

In a recent incident, a 16-year-old student faced 29 police reports for creating and sharing AI-generated pornographic deepfake images of other students from a private school in Malaysia. This case shows how simple it is to create false narratives and produce fake evidence. It also raises issues about the trustworthiness of forensic investigations and legal processes. As a result, law enforcement agencies urgently need dependable methods to verify the authenticity of media and lessen the negative effects of deepfakes.

Current deepfake detection methods often struggle to keep up with rapidly changing generation techniques. These methods also show limited effectiveness across different manipulation techniques and usually require a lot of computational power. This limits their practical use in forensic units and field investigations [5]. Forensic applications need lightweight, efficient, and reliable tools that can process media quickly and accurately without depending on high-performance hardware [5]. To meet this demand, Fakeseeker is introduced as a deepfake detection system built on the EfficientNet architecture. The goal is to design, implement, and evaluate Fakeseeker for analyzing image and video content to spot manipulated media. Key features include the ability to process uploaded files, provide real-time detection through camera and screen monitoring, and create detailed scan reports. The evaluation framework includes comparative testing of EfficientNet variants (B0, B1, B2) and assesses performance based on startup time, scan duration, and resource use. The evaluation ensures responsiveness and efficiency for real-time forensic work.

The remainder of this paper is organized as follows: Section 2 reviews related work in deepfake detection. Section 3 details the methodology, including the system architecture and model training pipeline. Section 4 discusses the experimental results. Finally, Section 5 concludes the study and describes future work.

2. Background of study

In this section, deepfake generation techniques are reviewed. Additionally, the previous study of detection models and details of the specific application strategy of EfficientNet within the Fakeseeker project are discussed. .

2.1 Deepfake Generation Techniques

Generative Adversarial Networks (GANs) [6] and Autoencoders [7] are foundational of deepfake generation techniques. GANs consist of two neural networks, the generator and the discriminator, that engage in adversarial training. It employs a competitive process between a generator that creates synthetic media and a discriminator that attempts to identify fakes, progressively improving realism. Sophisticated variants such as StyleGAN further enhance the quality and controllability of these synthetic outputs.

Autoencoders are a common technique for facial manipulation and data compression. An autoencoder has two main parts: an encoder that compresses an image into a latent representation and a decoder that reconstructs the image from this representation. Through this process, autoencoders learn to encode and reproduce facial features. This makes manipulations like face-swapping possible, where one person's features are encoded and then decoded onto another person.

Another technique is facial reenactment, which transfers expressions and movements from one face to another [8]. This method tracks the source's facial dynamics, maps them to key points on the target, and generates a new image with the transferred expressions. Early implementations, such as Face2Face, used this approach to replicate subtle facial movements, blinking, and lip synchronization, matching them with the source actor's performance [8].

2.2 Deepfake Detection Models

Recent research has explored various deepfake detection methods to enhance accuracy and address current limitations. A multimodal approach proposed in [9] combines a modified InceptionResNetV2 with Constant-Q Transform (CQT) for audio feature extraction and an improved XceptionNet for spatial analysis. This method shows the advantages of using different modules, reaching performance metrics of 97.52% accuracy and 98.04% F1-score. However, processing audio and visual data in parallel poses challenges for real-time use and scalability.

The Multiple Spatiotemporal Views Transformer (MSVT) introduced in [10] includes Local Spatiotemporal View (LSV) and Global Spatiotemporal View (GSV) to capture temporal dependencies in video sequences. Thheir findings show it performs well across six benchmark datasets, especially in temporal feature extraction. Similarly, [11] demonstrated that the TALL-Swin architecture achieves an AUC of 90.79% in cross-dataset testing by converting video clips into thumbnail layouts, maintaining spatial-temporal relationships. Despite these improvements, the complexity of transformer-based models still presents a major challenge for real-time use.

Hybrid architectures that combine CNNs and RNNs have also been studied for finding temporal inconsistencies in deepfake videos. For instance, [12] proposed the ResNet-Swish-BiLSTM framework. This framework processes sequential frames to identifyt artifacts and reaches an accuracy of 98.06%. However, this reliance on sequential processing causes latency, making it less suitable for real-time applications. To improve feature representation, transformer components have been added to detection pipelines. A cascaded network that combines EfficientNetV2S and Transformer achieved accuracies of 92.16% and 96.75% on the DFDC and

FaceForensics++ datasets [13]. Another method, StyleGRU [14], uses StyleGAN for facial attribute extraction and a style attention module to combine these with content features. While these models generate good performance results, their computational needs still hinder real-time use.

EfficientNet has become a promising alternative due to its lightweight design and good balance between accuracy and efficiency. Findings in [15] show that EfficientNet-B4 competes well with XceptionNet on datasets like FF++ and Celeb-DF. Further comparisons with Vision Transformers (ViT) in [16] highlight EfficientNet's suitability for environments with limited resources, providing faster inference and lower computational costs.

Most deepfake detection methods require a lot of computing power, which limits their use in the field and in real-time investigations. Digital forensics, on the other hand, often needs solutions that work well with limited hardware, like remote workstations or mobile devices. Systems that use transformers, hybrid networks, or multimodal designs usually take longer and use more resources. This scenario slowing down evidence analysis. To quickly verify if media is authentic without needing much preprocessing, forensic processes need tools that are efficient and easy to use. To fill this gap, therefore, the proposed system, Fakeseeker, uses EfficientNet-B0 to provide accurate deepfake detection in real time.

2.3 EfficientNet and Its Application Strategy in Fakeseeker

EfficientNet provides an effective solution for deepfake detection due to its innovative compound scaling method. This method increases the scale of the network depth, width, and resolution. As a result, EfficientNet models achieve higher accuracy with fewer parameters and lower computational cost (FLOPS) compared to other CNNs [17], [18]. This efficiency is important for real-time applications. Its ability to learn robust features at various scales makes it suitable at identifying the subtle, often low-level, artefacts characteristic of deepfakes [20]. The following set of equations governs the scaling:

- Depth (d): d = α^ϕ
- Width (w): w = β^ϕ
- Resolution (r): r = γ^ϕ

$$(\alpha \cdot \beta^2 \cdot \gamma^2)^{\phi} \approx 2^{\phi} \quad (1)$$

Here, α, β, and γ are constants that represent the scaling factors for depth, width, and resolution, respectively. These constants are determined by a small grid search done on the original baseline network. The purpose of this grid search is to find the best scaling ratios for the baseline architecture. This is undertaken while keeping the total increase in Floating Point Operations Per Second (FLOPs) is approximately (α · β^2 · γ^2)^ϕ ≈ 2^ϕ. The squared terms for β and γ reflect that computational cost (FLOPs) scales quadratically with network width and resolution, but linearly with depth. The compound coefficient ϕ is a user-specified parameter that controls the overall scale of the model. By increasing ϕ, all three dimensions (depth, width, and resolution) are uniformly scaled up according to the fixed ratios defined by α, β, and γ. For example, EfficientNet-B0 corresponds to ϕ=0 (conceptually, as the baseline), EfficientNet-B1 uses ϕ=1, EfficientNet-B2 uses ϕ=2, and so on, up to EfficientNet-B7. This scaling method promotes balanced growth in network depth, width, and input resolution as model capacity increases. This leads to better accuracy and efficiency compared to single-dimension or random scaling methods.

Fakeseeker's integration of EfficientNet uses a clear method for data processing and model adjustment. The system fine-tunes an EfficientNet model that was pre-trained on ImageNet [21]. This process includes changing the classifier head for binary classification (Real/Fake) and selectively unfreezing later convolutional layers. These layers include the final blocks, conv_head, bn1, and the classifier. The training for these layers uses different learning rates. This allows the model to recognize deepfake-specific patterns while keeping the general features from earlier frozen layers [17], [18]. Targeted fine-tuning along with proper data augmentation is key to reaching high detection accuracy.

3. Methodology

This section discusses the method used for designing, developing, and evaluating the Fakeseeker deepfake detection tool. It describes the system architecture, dataset preparation, model training pipeline, and algorithm design.

3.1 Fakeseeker System Architecture

Fakeseeker is built as a modular system that can manage various parts of deepfake detection. This includes acquiring media and showing the results. Fakeseeker supports both offline file-based scanning and live detection modes.

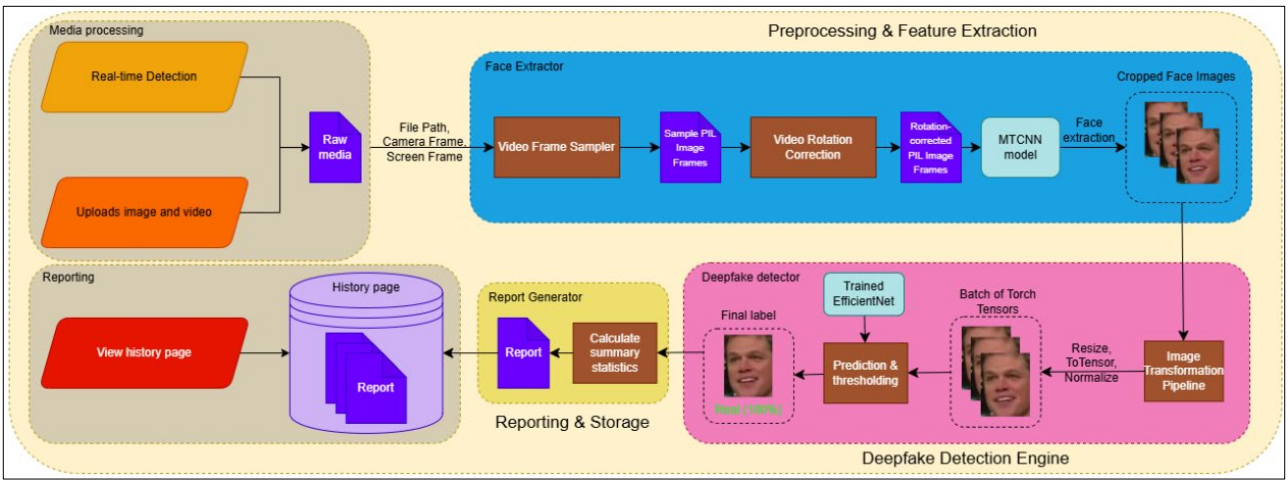


Fig. 1 *Fakeseeker architecture*

The main architecture (see Fig. 1) comprises several key modules as follows:

- Media Processing module handles users' uploads of images or videos and real-time feeds for camera or screen capture.
- Face Extractor module is responsible for detecting and isolating facial regions using Multi-task Cascaded Convolutional Networks (MTCNN).
- Deepfake Detector module uses the fine-tuned EfficientNet model for classification.
- Reporting and Storage module generates detailed scan summaries and managing scan history persistently in a user-specific data directory using JSON files and image thumbnails.
- User Interface (UI) built with Tkinter, providing different pages for home navigation, media upload, real-time detection, history viewing, and detailed report analysis.

The architecture can operate in two modes: image or video uploads and real-time detection. The tool begins with face detection and extraction, which uses MTCNN. When a face frame is selected, the tool scans the face frames using EfficientNet. After the scanning is completed, the tool generates a scanning result. A JSON file, which serves as the local database, stores all the results. To keep user data separate and persistent through application updates, this file and the generated thumbnails are stored in a specific user application data directory, rather than the application's installation folder. Users can view their scan results and take action through the Reporting module.

3.2 Model Training Pipeline

The core of Fakeseeker's detection capability lies in a fine-tuned EfficientNet model. The training pipeline (see Fig. 2) involves several key stages: Data Preparation & Preprocessing, Model initialisation and Fine-Tuning Setup, Evaluation and Artefact Storage.

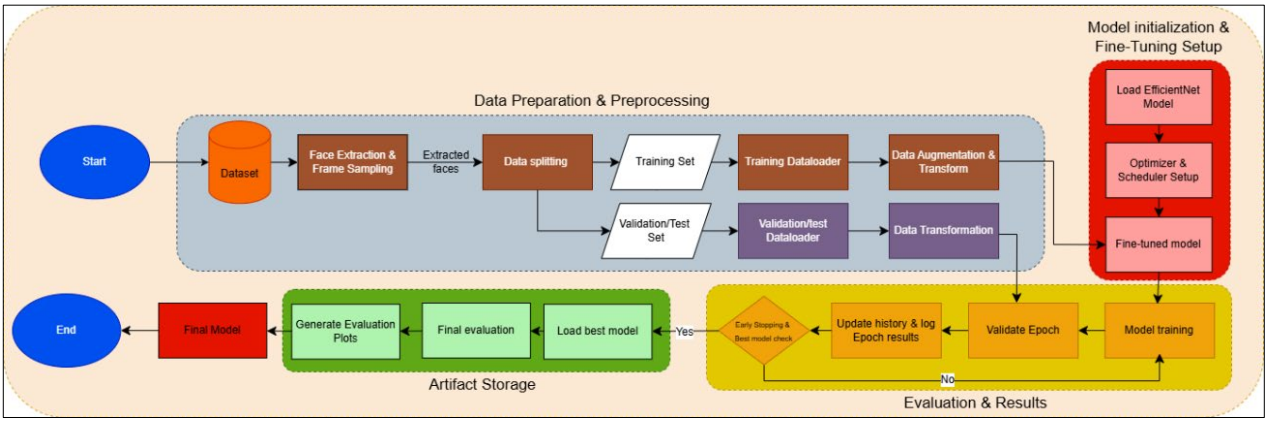


Fig. 2 *Model training process for deepfake detector*

J. of Soft Computing and Data Mining Vol. 7 No. 1 (2026) p. 37-54

The model training process starts with splitting the data into training and testing sets during preprocessing. The training set includes the part of the dataset used to train the AI model, which has input features and labels. Usually, the training set comprises most of the dataset. The testing or validation set is a separate dataset for evaluating the trained model's performance. It has input features similar to the training set, but the model has not seen these specific examples during training.

The ratio of the training set to the testing set is 80:20, which is a common split in machine learning. It means that 80 per cent of the dataset is allocated to the training set, while the remaining 20 per cent is used for the testing set. Then, face frames are extracted for both datasets and the EfficientNet model is trained. Subsequently, the model is evaluated and tested through a few iterations to develop the best model for the detection process.

3.3 Dataset Specification

In this study, the dataset is collected from publicly available and widely recognized deepfake datasets, including Celeb-DF (v1 and v2) [22], FaceForensics++ (FF++) [23], and the Deepfake Detection Challenge (DFDC) dataset [24]. This collection aimed to provide a broad range of deepfake generation techniques and real-world scenarios. It included approximately 6,444 fake videos, 72,898 fake images, and 728 real videos plus 20,636 real images before face extraction (see Table 1).

Table 1 *Composition of raw source datasets*

Dataset Name	Number of Sources (Fake)	Number of Sources (Real)
Celeb-DF v1	795 videos	408 videos
Celeb-DF v2	5639 videos	300 videos
FaceForensics++ (FF++)	40 videos	20 videos
DFDC	72898 images	20636 images
Subtotal:	6444 videos + 72898 images	728 videos + 20636 images

The composition of datasets evaluates the model's generalization to new samples from this combined distribution. Training the model on various datasets improves its ability to handle different deepfake techniques. This includes fully synthesized images and cross-dataset scenarios. This capability is important for forensic tools that must work well across various sources and formats of digital evidence.

3.4 Development Environment

FakeSeeker was developed and implemented in Python 3. The main framework for the graphical user interface was created using Python's standard Tkinter library and an improved design with themed widgets. Key libraries supporting the deepfake detection pipeline include PyTorch as the main deep learning framework. EfficientNet-PyTorch is used for the classification model architecture and Facenet-PyTorch is used for implementing the MTCNN face detector.

Image and video processing tasks, such as reading files, frame extraction, resizing, and converting color space, were mainly handled by OpenCV and Pillow(PIL). Pillow also facilitated image display within the Tkinter UI through ImageTk. Real-time screen monitoring relied on the MSS library for efficient screen capture. Platform-independent user data directory management was achieved using Appdirs, while background updates were handled with the Requests library. Standard libraries like os, json, logging, threading, and queue were essential for file operations, data persistence, logging, concurrent processing and communication between threads. The final standalone distribution application was created using PyInstaller.

3.5 Algorithm Design

Two key algorithms for Fakeseeker are the Train Model Module (see Fig. 3) and the Main Function Module (see Fig. 4). The Train Model Module focuses on preparing the deepfake detection model using EfficientNet architecture. The training process involves preprocessing the dataset. This step involves MTCNN face extraction, data augmentation, balanced sampling with WeightedRandomSampler, fine-tuning the pre-trained EfficientNet, unfreezing layers, applying different learning rates, AdamW optimizer, evaluating with Area Under the Curve (AUC) and generating report and confusion matrix plots. The preprocessing stage involves resizing images, normalizing pixel values, and isolating facial regions using face detection techniques. The dataset is split into training and validation subsets to allow for proper model evaluation. EfficientNet is modified for binary classification to differentiate between authentic and manipulated images. During training, model parameters are improved using backpropagation and a loss function such as binary cross-entropy, to achieve high detection accuracy. The final trained model is saved for use in the detection tool.

The main application module serves as the central controller for Fakeseeker. It manages user interactions and coordinates backend processes. It is built in Python using the Tkinter GUI framework, offering an easy-to-use interface for accessing key features like media upload, real-time detection, and viewing past scan results.

When the application starts, it sets up important components such as logging, the MTCNN-based face extraction system, and the fine-tuned EfficientNet detection model. It identifies user-specific data directories using the appdirs library to ensure the proper storage of scan history and generated thumbnails, based on the operating system's structure. The module then loads any existing scan records from a JSON file (scan_history.json) found in the designated user data directory. The application also features an automatic update check that runs in the background during startup. This check queries a specific URL for manifests that announce newer model versions available for download. After training the model, it will be exported for use in the main module for deepfake detection.

```
TRAIN_MODEL
1. BEGIN
2. INPUT: Raw dataset path (config['data_dir']), Configuration (config)
// Step 1: Data Preparation (Face Extraction - using FaceExtractor)
3. IF config['clean_start'] is True: DELETE processed_faces directory
4. CALL FaceExtractor.process_dataset(config['data_dir'], 'processed_faces', config['frames_per_video']) -> image_paths, labels
5. IF image_paths is empty: RAISE Error "No faces extracted"
// Step 2: Split Data
6. CALL train_test_split(image_paths, labels, test_size=config['val_split'], stratify=labels) -> train_paths, val_paths, train_labels, val_labels
// Step 3: Define Transforms & Datasets
7. DEFINE train_transform (Resize(config['image_size']), CenterCrop(config['image_size']), Augmentations[Flip, Rotate, Jitter, Blur?], ToTensor, Normalize(ImageNet))
8. DEFINE val_transform (Resize(config['image_size']), CenterCrop(config['image_size']), ToTensor, Normalize(ImageNet))
9. CREATE train_dataset = DeepfakeDataset(train_paths, train_labels, transform=train_transform)
10. CREATE val_dataset = DeepfakeDataset(val_paths, val_labels, transform=val_transform)
// Step 4: Create DataLoaders (Balanced Training)
11. CREATE train_loader using WeightedRandomSampler based on train_labels
12. CREATE val_loader (standard shuffle/no shuffle)
// Step 5: Load Model & Setup Fine-tuning
13. INITIALIZE model = EfficientNet.from_pretrained(config['model_version'], num_classes=2)
14. UNFREEZE last config['unfreeze_blocks'] blocks AND classifier (_fc, _conv_head, _bn1)
// Step 6: Define Loss & Optimizer (Differential LR)
15. SET criterion = CrossEntropyLoss() // OR FocalLoss() if implemented
16. DEFINE optimizer_grouped_parameters with differential learning rates (base, unfrozen, classifier) using config['lr'], config['classifier_lr_mult'], config['unfrozen_lr_mult']
17. SET optimizer = AdamW(optimizer_grouped_parameters)
18. SET scheduler = ReduceLROnPlateau(...)
// Step 7: Training Loop
19. INITIALIZE history, best_val_loss, early_stop_counter
20. FOR epoch = 1 TO config['epochs']:
21.   CALL train_epoch() -> train_loss, train_acc
22.   CALL validate() -> val_loss, val_acc
23.   UPDATE history
24.   LOG results
25.   CALL scheduler.step(val_loss)
26.   IF val_loss < best_val_loss:
27.     SAVE best_model checkpoint
28.     RESET early_stop_counter
29.   ELSE:
30.     INCREMENT early_stop_counter
31.   ENDIF
32.   IF early_stop_counter >= config['early_stopping_patience']: BREAK LOOP
33. ENDFOR
// Step 8: Post-Training Evaluation & Saving
34. SAVE final_model checkpoint
35. SAVE history (JSON), PLOT history (PNG)
36. LOAD best_model checkpoint
37. CALL evaluate_model() -> optimal_threshold, roc_auc
38. SAVE optimal_threshold.json (root and run_dir)
39. PLOT ROC curve (PNG)
40. CALL generate_evaluation_plots(optimal_threshold) -> saves Report/CM plots (PNG)
41. END
```

Fig. 3 *Algorithm of train model module*

Fig. 4 presents the main function module algorithm. This algorithm prepares the deepfake detection model using the EfficientNet architecture. The training process begins with preprocessing the dataset. This includes resizing images, normalizing pixel values, and isolating facial regions using face detection techniques. The dataset is split into training and validation subsets for effective model evaluation. EfficientNet is fine-tuned for binary

J. of Soft Computing and Data Mining Vol. 7 No. 1 (2026) p. 37-54

classification to tell apart real and manipulated images. The model uses an iterative training process to improve its parameters through backpropagation and a loss function such as binary cross-entropy. This helps it achieve high accuracy in detecting deepfakes. Finally, the trained model is saved for use in the detection tool.

```
MAIN_MODULE
1. BEGIN INITIALIZATION:
2.   Initialize Logger
3.   Determine User Data Paths (using Appdirs) -> userDataDir, reportsDir, thumbnailsDir, historyFile, userModelDir, userThresholdPath
4.   Create User Data Directories IF NOT EXIST
5.   Initialize UI Framework (Tkinter Root Window, Styles)
6.   Initialize Face Extractor (MTCNN)
7.   Initialize Deepfake Detector (EfficientNet, Load threshold/model prioritizing user paths then bundled defaults)
8.   Load Scan History from historyFile (JSON in user data dir)
9.   Initialize UI Pages (Home, Upload, Realtime, History, Report, Sidebar, Toolbar) & Layout
10. START Background Update Check (Threaded)
11. DISPLAY Home Page
12. END INITIALIZATION
13. BEGIN MAIN EVENT LOOP (Tkinter mainloop)
// --- Event Handlers / User Actions ---
14. FUNCTION HANDLE_UPLOAD_BUTTON_CLICK:
15.   CALL fileDialog.askopenfilename() -> selectedFile
16.   IF selectedFile exists:
17.     Update Upload Page Preview
18.     Enable Scan Button
19.   ELSE:
20.     Clear Upload Page Preview
21.     Disable Scan Button
22.   ENDIF
23. ENDFUNCTION
24. FUNCTION HANDLE_SCAN_BUTTON_CLICK(filePath):
25.   IF filePath is NULL: Show Warning "No file selected" & RETURN
26.   Disable Scan Button
27.   Update Status Label "Starting scan..."
28.   START Background Thread -> CALL PROCESS_MEDIA_ASYNC(filePath)
29.   SCHEDULE CHECK_SCAN_QUEUE() // Start polling queue
30. ENDFUNCTION
31. FUNCTION PROCESS_MEDIA_ASYNC(filePath): // Runs in Background Thread
32.   PUT status "Extracting faces..." ON scanQueue
33.   CALL FaceExtractor -> faces[]
34.   IF faces is empty: PUT error "No faces detected" ON scanQueue & GOTO Cleanup
35.   PUT status "Saving thumbnails..." ON scanQueue
36.   CREATE scanDir in thumbnailsDir (user data)
37.   savedThumbPaths[] = []
38.   validFaces[] = []
39.   FOR each face in faces[]:
40.     SAVE face to scanDir -> absThumbPath
41.     APPEND absThumbPath to savedThumbPaths[]
42.     APPEND face to validFaces[]
43.   ENDFOR
// Helper Function: Real-Time Detection
38. FUNCTION REALTIME_DETECTION:
// Called by main module Case 2
39.   // **Note: Actual implementation uses threading**
40.   INITIALIZE rt_results_list[], rt_counts, rt_last_store_time
41.   WHILE media_stream (camera/screen) is active AND detection_active:
42.     Capture current frame
43.     CALL FaceExtractor.extract_faces_and_boxes_realtime(frame) -> faces[], boxes[]
44.     FOR each face_pil, box in faces[], boxes[]:
45.       prediction = CALL Detector.predict_pil(face_pil) -> (label, probability) or None
46.       IF prediction is valid:
47.         DISPLAY box and prediction result on feed
48.         IF time_since(rt_last_store_time) >= rt_store_interval:
49.           SAVE face_pil thumbnail -> abs_thumb_path (user thumbnails dir)
50.           APPEND {prob: probability, thumb_path: abs_thumb_path} to rt_results_list[]
51.           UPDATE rt_last_store_time, rt_counts
52.         ENDIF
53.       ENDIF
54.     ENDFOR
55.   ENDWHILE
56.   // **Actual code does this in stop_detection**
57.   IF rt_results_list is not empty:
58.     CALCULATE summary (avg_prob, overall_label, counts) from rt_results_list[]
59.     CREATE summary_report dictionary {timestamp, summary{}, results[avg_prob], face_thumbnails[abs_paths], type='real-time-summary'}
60.     CALL update_scan_history(summary_report) // Saves summary to JSON
61.   ENDIF
62. ENDFUNCTION
```

Fig. 4 *Algorithm of main function module*

In the main module, users can upload media files for analysis, conduct real-time detection, or access previously generated reports. Depending on the user's choice, the module processes input media, applies the trained EfficientNet model to detect manipulations, and displays results. This module ensures smooth functionality integration while keeping the user experience easy to navigate.

The media processing module allows users to select files through a standard dialogue interface. After selecting a file, a preview of the chosen media appears. Analysis starts when the user activates the "Start Scan" option. The processing workflow includes face extraction using MTCNN, storing thumbnails in the user's data folder, and generating predictions for each detected face with EfficientNet. These tasks run in a separate background thread to keep the interface responsive. Status updates are sent to the main UI thread through a thread-safe queue and displayed to the user. Once everything is finished, the results, which include probability scores, scan type, and paths to saved thumbnails, are returned through the same queue. The application then automatically navigates to display the Detailed Report Page for that scan. The user is prompted to save this scan result permanently to the history JSON file.

This real-time detection module allows live detection of deepfakes by analyzing video streams frame by frame. The user can choose to activate either the system camera or screen monitoring. Activating screen monitoring hides the main application window and shows a minimal floating toolbar with controls like Start/Stop Detection, Show Status, Show Main Window. The selected feed is processed frame-by-frame. If "Start Detection" is activated, MTCNN extracts faces, while EfficientNet checks their authenticity in real-time. Results are limited to manage performance and the size of the history, typically saved once per second for each detected face. When the user stops detection, closes the toolbar, or navigates away, a summary report for the whole session is created. The report includes total counts, average fake probability, and a list of all saved thumbnail paths. It is then added to the main *scan_history.json* file.

4. Results and Discussion

This section presents the implementation details of Fakeseeker, including model training implementation, followed by the experimental results from the model's training and evaluation, key user interface elements and the tool's performance evaluation.

4.1 Model Training Implementation

This section describes the pipeline for training the EfficientNet deepfake detection model. It covers the initial setup to the final evaluation and presents the key results.

4.1.1 Dataset Preparation

The first preprocessing step is to extract faces from the raw media. The FaceExtractor class works on each image or video frame. For videos, a set number of frames is sampled to capture changes over time. For videos, a specific number of frames is sampled to capture changes over time. Detected faces are filtered based on minimum confidence and size thresholds to ensure quality. All successfully extracted faces are saved as separate JPEG images.

This process resulted in a large dataset of 200,925 processed faces (see Table 2). This organized list includes face image paths and their binary labels (0 for real, 1 for fake). The data was then split into training and a combined test set using stratified sampling to maintain consistent class distribution.

Table 2 *Processed face dataset composition*

Category	Number of extracted faces	Percentage of total extracted faces
Real faces	20393	10.15%
Fake faces	180532	89.85%
Total Faces	200,925	100%

The high precision and recall for both 'Real' and 'Fake' classes (see Fig. 5, Fig. 6, Fig. 7) confirm that the chosen architecture, data preprocessing, conservative augmentation strategy, and fine-tuning approach were effective. This strategy helps the model to learn important features within this data domain.

4.1.2 Data Transformation and Augmentation

These changes increase the variety of the training data, exposing the model to different conditions and reducing its chance of overfitting. It helps the model generalize better to new deepfakes it may encounter in real-world scenarios. The Transformation and Augmentation steps are defined within the *trainer._get_transforms()* method.

J. of Soft Computing and Data Mining Vol. 7 No. 1 (2026) p. 37-54

All images, whether for training or validation, are resized by first scaling the shorter edge to *config['image_size']*, followed by centre-cropping to a square of *config['image_size'] x config['image_size'])*. They also undergo normalisation using the standard ImageNet mean and standard deviation values. Normalization ensures that the input data distribution matches what the pre-trained EfficientNet model expects. The training dataset receives extra random augmentations: horizontal flipping, minor rotations, color jitter, and potentially Gaussian blurring.

4.1.3 Model Loading and Fine-tuning Setup

An EfficientNet-B2 model was loaded using EfficientNet.from_pretrained('efficientnet-b2', num_classes=2). It used weights that were pre-trained on ImageNet. The fine-tuning strategy included making specific layers trainable. The final fully connected layer (_fc) was always set to be trainable. According to config['unfreeze_blocks'] (default: 3), the last three convolutional blocks of the EfficientNet architecture, along with the _conv_head and _bn1 layers before the classifier, were unfrozen. This allowed their weights to be updated during training. Layers earlier in the network stayed frozen, which kept their learned ImageNet features intact.

The model was then moved to the right Compute Unified Device Architecture (CUDA). It used a CUDA GPU if one was available; otherwise, it used the CPU. This method keeps the learned ImageNet features and adjusts the deeper layers and the classifier for the deepfake task. It leads to faster convergence and better performance than training from scratch or fine-tuning the whole network.

4.2 Model Performance Evaluation

The training efficiency was measured using total training time, average time per epoch, the number of epochs to achieve the best test loss, and peak GPU memory usage (see Table 3). Model effectiveness was evaluated on test sets by looking at accuracy, AUC, log loss, precision, recall, F1-score, balanced accuracy, specificity, and confusion matrices (see Tables 4, 5, and 6). Youden's J statistic was used to determine the best classification threshold that balanced sensitivity and specificity.

Table 3 *Training efficiency comparison*

Metric	EfficientNet-B0	EfficientNet-B1	EfficientNet-B2
Total Train Time (HH:MM: SS)	09:08:51	09:54:34	12:41:45
Avg. Time/Epoch (MM: SS)	~08:10 - 10:00 (avg. ~9m)	~11:50 - 12:05 (avg. ~11m 53s)	~15:07 - 16:23 (avg. ~15m 14s)
Epochs to Best Val/Test Loss	49 (Val Loss: 0.0021)	39 (Val Loss: 0.0036)	37 (Val Loss: 0.0029) / 40 (Val Loss: 0.0029)
Peak GPU Memory (GB)	0.3 – 0.5	0.3 – 0.6	0.5 – 0.8

Table 4 *Model effectiveness comparison on test set*

Metric	EfficientNet-B0	EfficientNet-B1	EfficientNet-B2
Accuracy (%)	99.2%	98.8%	99.1%
AUC	0.9996	0.9993	0.9994
LogLoss (Best)	0.0021	~0.0035	~0.0029
Balanced Acc (%)	0.9906	0.9875	0.9881

Table 5 *Model effectiveness comparison on test set (fake classes)*

Metric	EfficientNet-B0	EfficientNet-B1	EfficientNet-B2
Precision	0.97	0.93	0.98
Recall	0.99	0.99	0.98
F1-Score	0.98	0.96	0.98

Table 6 *Model effectiveness comparison on test set (real classes)*

Metric	EfficientNet-B0	EfficientNet-B1	EfficientNet-B2
Precision	1.00	1.00	1.00
Recall	1.00	0.99	1.00
F1-Score	1.00	0.99	1.00

All three EfficientNet variants, B0, B1, and B2, showed successful learning patterns (see Fig. 5, Fig. 6, Fig. 7). The training loss decreased steadily for all models. At the same time, training accuracy often exceeds 95%. Test loss showed very low values before leveling off, along with high validation and test accuracies. The difference between training and test metrics was usually small. The results show that data augmentation together with targeted fine-tuning offered effective regularization. Early stopping techniques also helped reduce overfitting. Among the variants tested, EfficientNet-B0 and EfficientNet-B2 had slightly lower validation loss than EfficientNet-B1.

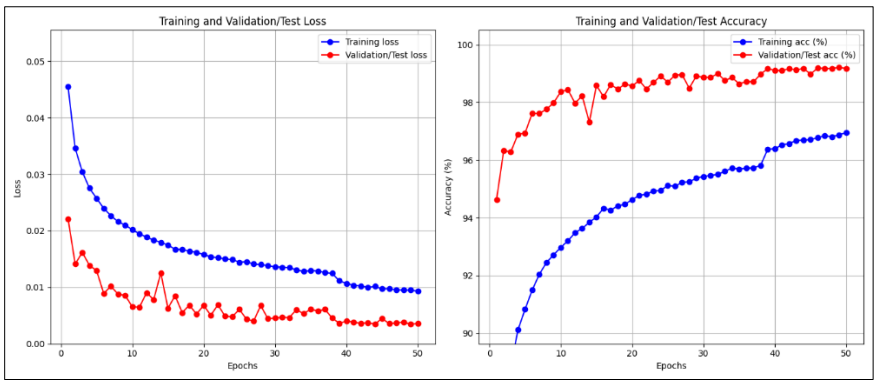


Fig. 5 *Learning curves for EfficientNet-B0*

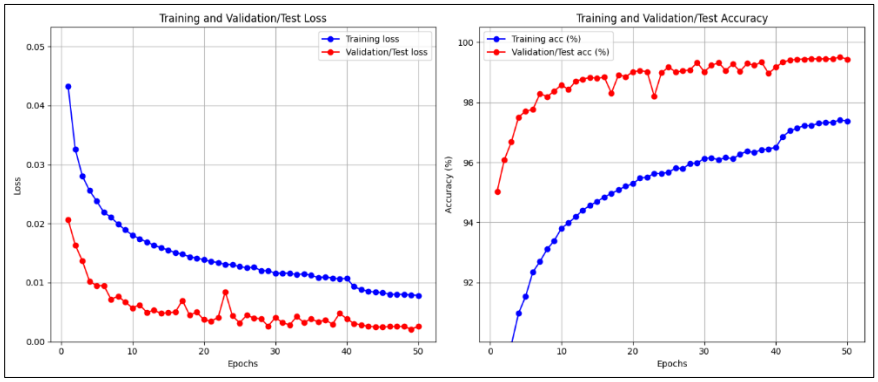


Fig. 6 *Learning curves for EfficientNet-B1*

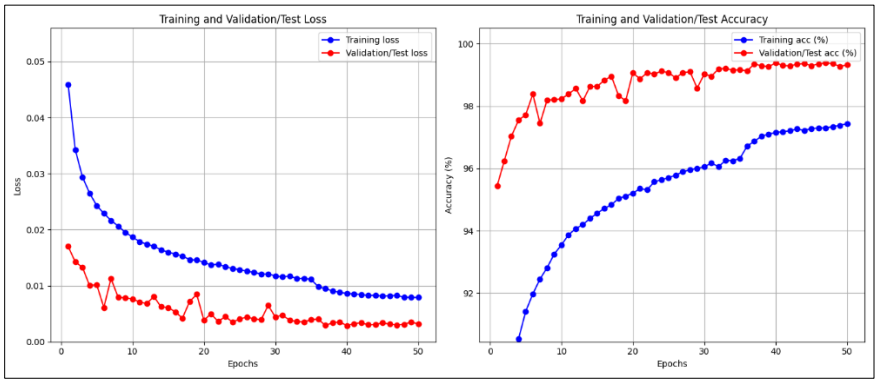


Fig. 7 *Learning curves for EfficientNet-B2*

J. of Soft Computing and Data Mining Vol. 7 No. 1 (2026) p. 37-54

All three tested EfficientNet variants (B0, B1, B2) delivered high performance on the FakeSeeker test dataset. Regarding training efficiency, EfficientNet-B0 was the fastest to train per epoch, averaging about 8 to 9 minutes. EfficientNet-B1 followed with about 12 minutes, and EfficientNet-B2 took around 15 minutes. These results match their model sizes and input image resolutions.

All models achieved outstanding AUC scores with B0: 0.9996, B1: 0.9993, B2: 0.9994. It further indicates that they can effectively distinguish between classes at all thresholds. EfficientNet-B0 achieved the lowest best validation LogLoss (0.0021), followed closely by EfficientNet-B2 (0.0029). This suggests the model made highly confident and accurate predictions. Precision for the 'Fake' class was perfect (1.00) for all models, which means no 'Real' images were misclassified when the model predicted 'Fake' at its best threshold. Recall for 'Fake' was also very high (0.99-1.00). For the 'Real' class, EfficientNet-B0 and EfficientNet-B2 showed slightly better precision between 0.97 and 0.98 compared to EfficientNet-B1 which has a precision of 0.93). All models, however, maintained very high recall between 0.98 and 0.99.

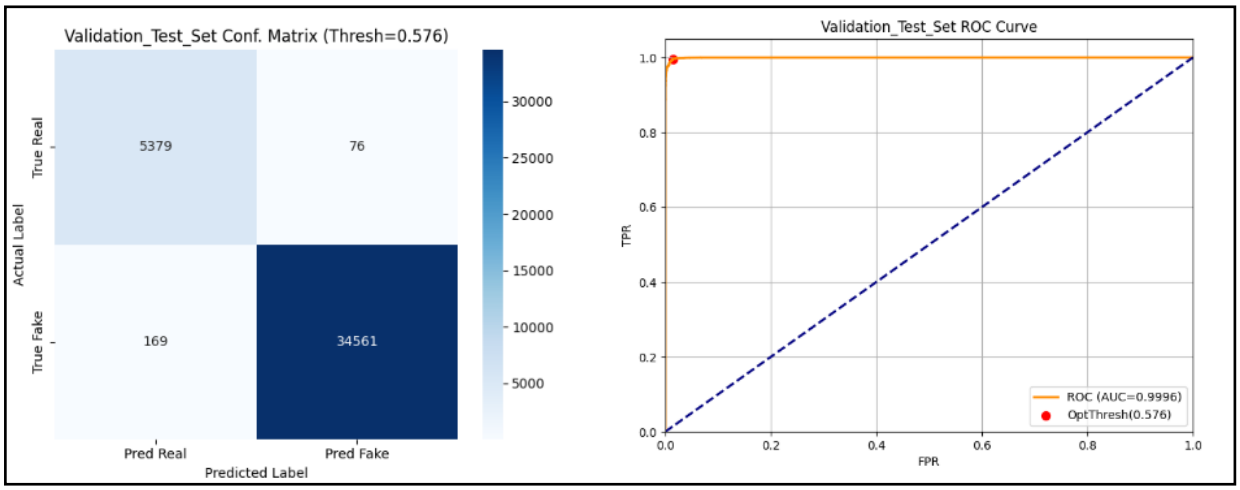


Fig. 8 *Evaluation plots for EfficientNet-B0*

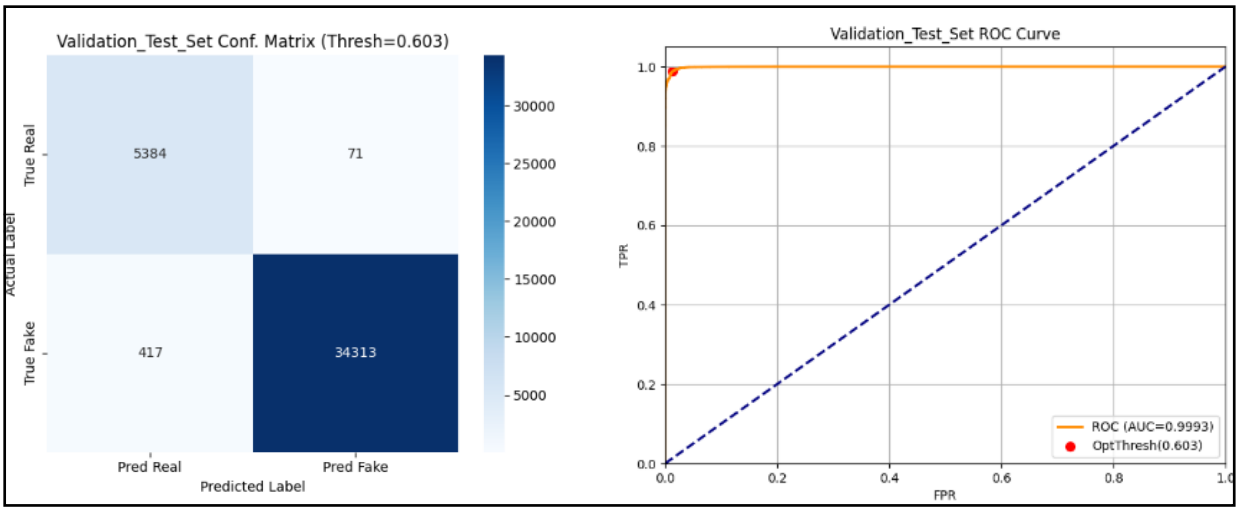


Fig. 9 *Evaluation plots for EfficientNet-B1*

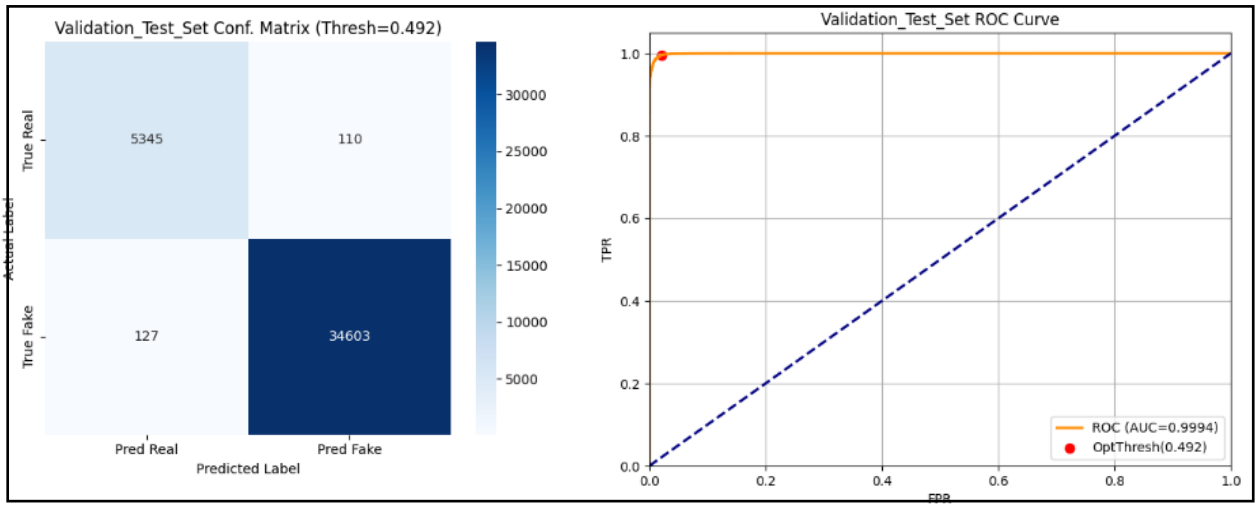


Fig. 10 *Evaluation plots for EfficientNet-B2*

In this evaluation, all three EfficientNet variants—B0, B1, and B2—performed exceptionally high. They all had testing accuracies over 98.7% and AUC scores above 0.999. Among these, EfficientNet-B0 demonstrated the best balance between detection accuracy and computational efficiency, making it the best choice for Fakeseeker. Although EfficientNet-B2 achieved a slightly higher AUC in one run, it was the most computationally intensive, requiring about 15 minutes per training epoch. EfficientNet-B1 offered a reasonable middle ground, with an average of 12 minutes per epoch and solid results. On the other hand, EfficientNet-B0 was the fastest to train at 8-9 minutes per epoch and achieved the lowest validation LogLoss of 0.0021, indicating highly confident predictions. It also maintained perfect precision of 1.00 for the 'Fake' class and high precision for the 'Real' class (0.97–0.98), outperforming B1.

Given these results, EfficientNet-B0 was selected as the backbone model for Fakeseeker. It achieved a top checkpoint testing accuracy of about 99.2% and an AUC of 0.9996 using an optimal threshold of 0.576.

4.3 Comparative Analysis of Prior Work

To understand the performance of the EfficientNet models developed for FakeSeeker, a comparison was made with a recent study by Singh et al. [25]. The study also evaluated different EfficientNet variants (B0-B7) for deepfake detection (see Table 7). It used a dataset of approximately 1000 images from Kaggle, split into an 80% training set and a 20% test set.

Table 7 *Comparative analysis of FakeSeeker's EfficientNet models with prior work*

Metric	Model Variant	Fakeseeker	Singh et al. [25]
Accuracy (%)	EfficientNet-B0	99.2	76.3
	EfficientNet-B1	98.8	78.5
	EfficientNet-B2	99.1	77.2
Precision	EfficientNet-B0	0.984	0.616
	EfficientNet-B1	0.963	0.716
	EfficientNet-B2	0.987	0.646
Recall	EfficientNet-B0	0.991	0.872
	EfficientNet-B1	0.987	0.830
	EfficientNet-B2	0.988	0.862
F1-Score	EfficientNet-B0	0.987	0.722
	EfficientNet-B1	0.975	0.769
	EfficientNet-B2	0.987	0.738
Dataset size	All	~160740 Train, ~40185 test	~800 Train, ~200 Test
Key Different	All	Fine-tuned on curated FF++, CelebDF, DFDC faces; Specific augmentations; Early stopping	Kaggle dataset; Details on specific preprocessing, augmentation is limited

J. of Soft Computing and Data Mining Vol. 7 No. 1 (2026) p. 37-54

Table 7 presents that FakeSeeker's EfficientNet-B0 model achieved a test accuracy of 99.2%, significantly outperforming the 76.3% accuracy reported for EfficientNet-B0 by Singh et al. [25]. Similarly, FakeSeeker's EfficientNet-B2, with an accuracy of 98.8%, also demonstrated superior accuracy compared to 77.2% in the prior study. For EfficientNet-B1, FakeSeeker achieved 99.1%, higher than the 78.5% reported by Singh et al., who identified B1 as their best-performing variant.

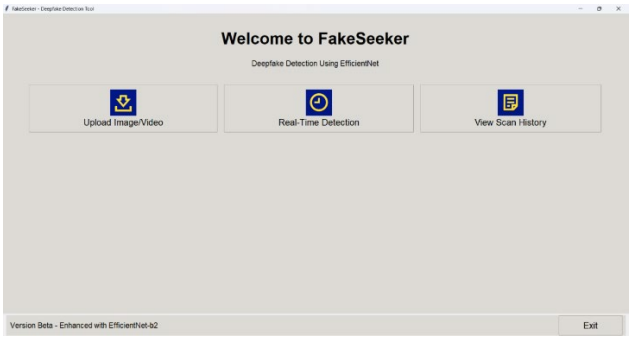
A likely explanation for the observed performance variations is that FakeSeeker was trained and evaluated on a significantly larger and more diverse curated dataset comprising 200,925 processed faces derived from multiple sources, including FaceForensics++, Celeb-DF, and DFDC. This contrasts with the ~1000 images used in the study by [25]. Larger and more diverse datasets generally enable models to learn more robust and generalizable features, which is a likely factor in the higher performance metrics observed for FakeSeeker's models.

Fakeseeker uses a dedicated pipeline for face extraction with MTCNN. It also uses a specific set of data augmentation methods during training. In comparison, Singh et al. [25] mention normalization and resizing but give limited details about their data preparation and augmentation methods, making it hard to compare directly. Data augmentation is important for improving how well the model performs on new data. Fakeseeker's training process includes fine-tuning pre-trained EfficientNet models with specific settings, such as selectively unfreezing blocks, using different learning rates, and implementing early stopping. For example, EfficientNet-B0 achieved the lowest validation loss at epoch 49, as shown in Table 5.8. Furthermore, FocalLoss was used to handle class imbalance during training. While earlier studies mention fine-tuning, they provide fewer details on unfreezing methods or optimization techniques beyond the usual practices.

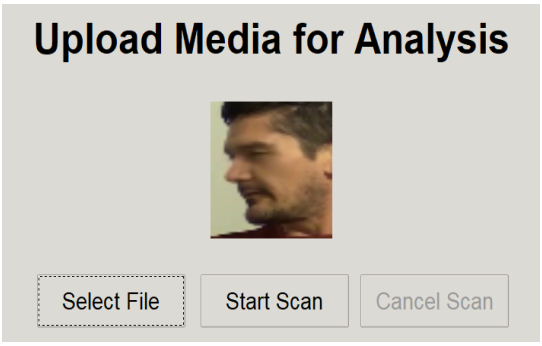
Overall, both studies are consistent that EfficientNet architectures are suitable for deepfake detection. However, FakeSeeker's models, particularly EfficientNet-B0, show better performance in terms of accuracy and training efficiency on their test sets. This is likely due to the larger and more diverse dataset, along with the specific fine-tuning and data augmentation strategies used. The earlier study found that EfficientNet-B1 was the best among their tested variants. This contrasts with FakeSeeker's outcome where EfficientNet-B0 provided the best balance. It also highlights the sensitivity of model performance to dataset characteristics and training setup.

4.4 A Snapshot of Fakeseeker Tool

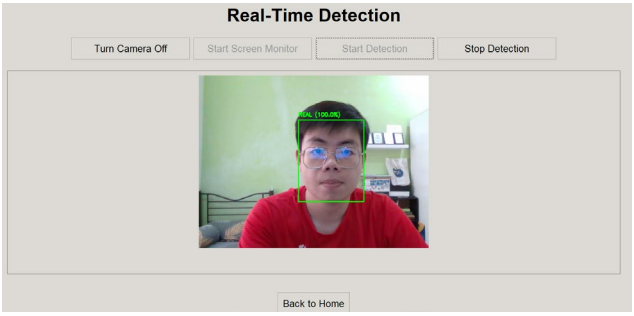
This section demonstrates that Fakeseeker has been successfully designed, developed, and tested using EfficientNet-B0 as the backbone. Figure 11(a) presents the main Home Page, which gives access to the main functionalities. Users can choose to upload media for analysis in Figure 11(b), start real-time detection in Figure 11(c) & (d), or view past results in Figure 11(e). Detailed scan reports are presented as shown in Figures 11(f) and 11(g).



(a)



(b)



(c)

(d)

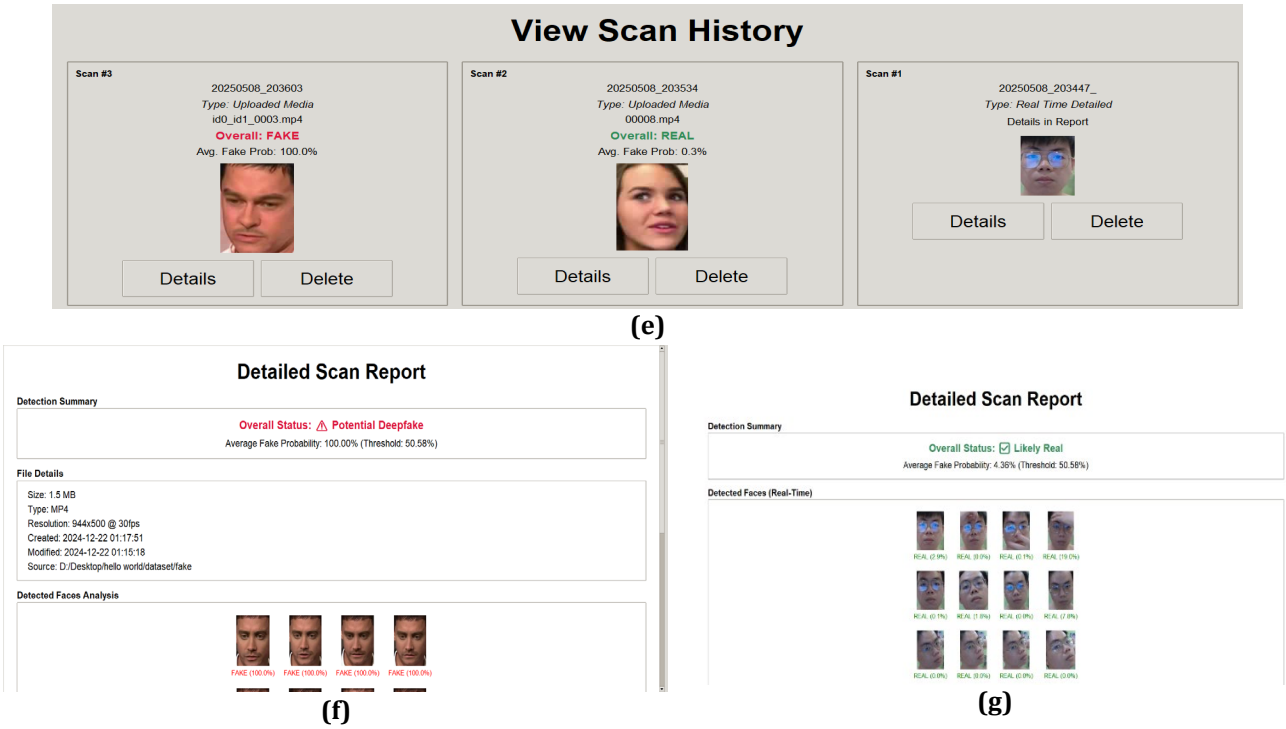


Fig. 11 *User Interface (a) Home page; (b) Upload page; (c) Real-time page (Camera); (d) Real-time page (Screen); (e) History page; (f) Example of fake result; (g) Example of real result*

The Fakeseeker interface generates a detailed scan report that is essential for examining digital media. The report includes metadata like file size, format, resolution, creation and modification timestamps, source path, and deepfake detection results. This metadata provides important context about the media's origin and history. It helps investigators find anomalies or possible signs of tampering [26].

The detection summary section presents the classification status, including “Potential Deepfake,” the average fake probability, and threshold values. The face analysis section displays detected faces along with confidence scores. This information enables forensic analysts to verify the media's authenticity, investigate potential manipulation patterns, and maintain a clear chain of custody. As Ticovan and Gheorghe point out [27], forensic tools need to detect anomalies and also preserve metadata and context to ensure it can be used in legal cases.

4.4.1 Fakeseeker Performance Evaluation

The performance of the FakeSeeker application was evaluated including scan time and memory usage. Table 8 summarizes the test plan and results.

J. of Soft Computing and Data Mining Vol. 7 No. 1 (2026) p. 37-54

Table 8 *Performance testing plan for the application*

No.	Test list	Description	Expected Result	Actual Result
1	Performance Test	Measure application startup time (from launch to Home Page visible)	Startup time is reasonable (<10 seconds, threshold defined)	Pass
		Measure scan time for a small image, a large image, a short video, and a long video.	Scan times are acceptable relative to file size and type. Status updates are timely. UI remains responsive	Pass
		Monitor CPU/Memory usage during idle, file scan, real-time camera during idle, real-time camera during detection, real-time screen during detection	Usage remains within acceptable limits. No excessive spikes or memory leaks. CPU usage drops when idle	Pass
		Test the responsiveness of scrolling on History/Report pages with 10-50 entries.	Scrolling remains smooth without significant lag	Pass

The application startup time, measured from launch to when the Home Page is fully visible and interactive, was consistently under 10 seconds on the test system described in the hardware specifications. It is well within the acceptable range of under 10 seconds (see Table 9). Scan times were acceptable and varied based on file size and type. A small image was processed in approximately 2 seconds. A larger image took around 3 seconds. A short video was processed in approximately 6 seconds. A longer video took approximately 10 seconds. A longer, larger video took about 15 seconds.

Table 9 *Performance testing results for scan time*

Image description	Time process
Small image (31.6 KB, 160 × 160 resolution, PNG File (png))	~2 seconds
Large image (3.4 MB, 3000 × 4000 resolution, JPG File (.jpg))	~3 seconds
Short video (1.51 MB, 11s, 944 × 500 resolution, 30 frame rate, MP4 File (.mp4))	~6 seconds
Longer video (11.8 MB, 67s, 360 × 640 resolution, 57.939 Frame rate, MP4 File (.mp4))	~10 seconds
Longer and larger video (149 MB, 53s, 1920 × 1080 resolution, 57.939 Frame rate, MP4 File (.mp4))	~15 seconds

Table 10 *Performance testing results for memory usage*

Application status	Memory usage (CPU)
Background (Word and Task Manager were running)	4%-6% Utilization, 1.30GHz to 1.60GHz Speed
In idle	4%-10% Utilization, 1.30GHz to 1.72GHz Speed
During a file scan	11%-18% Utilization, 1.86GHz to 2.90GHz Speed
Real-time camera during idle	11%-18% Utilization, 1.86GHz to 3.10GHz Speed
Real-time camera during detection	22%-30% Utilization, 2.44GHz to 3.10GHz Speed
Real-time screen during detection	16%-30% Utilization, 2.42GHz to 3.10GHz Speed

System resource use was monitored during different application states (see Table 10). When idle, CPU usage was low, approximately 4% to 6%. During image or video scans, CPU usage spiked as expected, from 4% to 10% for processing and model inference. However, it returned to low levels after completion. Memory usage has temporarily increased, but no leaks were detected. With the camera feed on but detection off, CPU usage was moderate between 11% to 18% due to video capture and display. During active real-time detection, CPU usage was between 22% and 30%. Similarly, screen monitoring with active detection caused a significant increase in CPU usage.

Table 11 *Performance testing results for the responsiveness of the scrolling page in the history page*

Number of reports	UI Responsiveness	Observations
10	Instant	No issues
20	Instant	No issues
30	Instant	The scrolling page becomes less smooth
40	Instant	Scrolling page becomes less smooth
50	Take a second to show	Increase in load time, the page is unstable when changing pages

The responsiveness of scrolling in the History and Report pages, especially with many entries, was evaluated (see Table 11). Scrolling was smooth, but there was noticeable lag when the number of reports reached 50. This was due to loading many entries into the application.

The performance evaluation of Fakeseeker demonstrates an efficient, lightweight forensic tool. Application initialization consistently finished in under 10 seconds. Scan times varied from 2 to 15 seconds, depending on the media size and type. Resource utilization remained low to moderate, with CPU usage reaching about 30% during active real-time detection. Memory use was steady throughout the process, with no signs of leaks. These results match what was observed in [28], which highlights that optimization is a key factor in the performance of digital forensic tools, along with standardization and systematic evaluation. Furthermore, it is important to maintain a balance between efficiency, transparency, and ethical implementation for AI-driven forensic solutions. This balance helps build trust and protects the integrity of investigative processes [29], [30].

5. Conclusion

This research presents the design, implementation, and evaluation of Fakeseeker, a deepfake detection tool that supports digital forensic workflows and improves evidence analysis. The key contributions of this work include:

- An innovative method for real-time deepfake detection.
- An assessment of the method's effectiveness using data augmentation and related optimization techniques.
- On-site verification of media authenticity in resource-limited environments without compromising detection reliability. The system uses a modular pipeline that goes from media ingestion to reporting, ensuring traceability and documentation, which are essential for forensic applications.

Although Fakeseeker demonstrates good performance, several improvements are suggested to address current limitations. Creating a web-based version would make it more accessible and allow for URL-based analysis, removing the need for local installation. Further improvements should aim to refine the face extraction process to handle non-standard video orientations, like rotated frames. Regularly retraining the model with varied and updated datasets, including new deepfake creation methods, is crucial to keep detection reliable.

Acknowledgement

This research was supported by Universiti Tun Hussein Onn Malaysia (UTHM) through Tier 1 (Vot Q931).

Conflict of Interest

Authors declare that there is no conflict of interest regarding the publication of the paper.

Author Contribution

The authors confirm contribution to the paper as follows: ***study conception and design:*** *Chua Kai Zen, Nurul Hidayah Ab Rahman;* ***data collection:*** *Chua Kai Zen;* ***analysis and interpretation of results:*** *Chua Kai Zen;* ***draft manuscript preparation:*** *Chua Kai Zen, Nurul Hidayah Ab Rahman, Niken Dwi Wahyu Cahyani, Mubarak Saif Mohsen. All authors reviewed the results and approved the final version of the manuscript.*

Declaration of AI Use in Manuscript Preparation

During the preparation of this work, the author(s) used Grammarly and Microsoft CoPilot in order to check for grammatical errors and improve the sentences. After using the tools, the author(s) reviewed and edited the content as needed and took full responsibility for the content of the published article.

References

[1] Negi, S., Jayachandran, M., & Upadhyay, S. (2021). Deep fake: An understanding of fake images and videos. *International Journal of Scientific Research in Computer Science Engineering and Information Technology, 7*(3), 183–189. https://doi.org/10.32628/CSEIT217334

[2] Garg, D., & Gill, R. (2023, December). Deepfake generation and detection – An exploratory study. In *Proceedings of the 2023 10th IEEE Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON)* (pp. 888–893). IEEE. https://doi.org/10.1109/UPCON59197.2023.10434896

[3] Tan, B. (2025, April 15). Johor police: Seven more reports lodged against teen who generated deepfake photos of Kulai students. *Malay Mail.* https://www.malaymail.com/news/malaysia/2025/04/15/johor-police-seven-more-reports-lodged-against-teen-who-generated-deepfake-photos-of-kulai-students/173180

[4] Chesney, R., & Citron, D. K. (2019). Deep fakes: A looming challenge for privacy, democracy, and national security. *California Law Review, 107*(6), 1753–1804.

[5] Verdoliva, L. (2020). Media forensics and DeepFakes: An overview. *IEEE Journal of Selected Topics in Signal Processing, 14*(5), 910–932. https://doi.org/10.1109/JSTSP.2020.3002101

[6] Karras, T., Laine, S., & Aila, T. (2021). A style-based generator architecture for generative adversarial networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 43*(12), 4217–4228. https://doi.org/10.1109/TPAMI.2020.2970919

[7] Das, M. K., Kumar, M., Kapil, I. K., & Yadav, R. K. (2023, May). Deepfake creation using GANs and autoencoder and deepfake detection. In *Proceedings of the 2023 2nd International Conference on Vision Towards Emerging Trends in Communication and Networking Technology (ViTECoN)* (pp. 1–6). IEEE. https://doi.org/10.1109/ViTECoN58111.2023.10157962

[8] Megahed, A., & Han, Q. (2020, December). Face2Face manipulation detection based on histogram of oriented gradients. In *Proceedings of the 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (pp. 1260–1267). IEEE. https://doi.org/10.1109/TrustCom50675.2020.00169

[9] Elpeltagy, M., Ismail, A., Zaki, M. S., & Eldahshan, K. (2023). A novel smart deepfake video detection system. *International Journal of Advanced Computer Science and Applications, 14*(1).

[10] Yu, Y., Zhang, Y., Liu, Y., Du, J., Wang, J., & Zhou, J. (2023). MSVT: Multiple spatiotemporal views transformer for deepfake video detection. *IEEE Transactions on Circuits and Systems for Video Technology, 33*(9), 4462–4471. https://doi.org/10.1109/TCSVT.2023.3281448

[11] Xu, Y., Liang, J., Jia, G., Yang, Z., Zhang, Y., & He, R. (2023). TALL: Thumbnail layout for deepfake video detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 22658–22668).

[12] Qadir, A., Mahum, R., El-Meligy, M. A., Ragab, A. E., AlSalman, A., & Awais, M. (2024). An efficient deepfake video detection using robust deep learning. *Heliyon, 10*(5).

[13] Deng, L., Wang, J., & Liu, Z. (2023). Cascaded network based on EfficientNet and transformer for deepfake video detection. *Neural Processing Letters, 55*(6), 7057–7076.

[14] Choi, J., Kim, T., Jeong, Y., Baek, S., & Choi, J. (2024). Exploiting style latent flows for generalizing deepfake video detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 1133–1143).

[15] Yasser, B., Hani, J., El-Gayar, S., Amgad, O., Ahmed, N., Ebied, H. M., Amr, H., & Salah, M. (2023). Deepfake detection using EfficientNet and XceptionNet. In *2023 Eleventh International Conference on Intelligent Computing and Information Systems (ICICIS)* (pp. 598–603). IEEE.

[16] Singh, S., Sarala, P., Chandra, S. K., & Kumar, M. D. (2024). DeepFake image detection and classification using EfficientNet model. In *2024 OPJU International Technology Conference (OTCON) on Smart Computing for Innovation and Advancement in Industry 4.0* (pp. 1–5). IEEE. https://doi.org/10.1109/OTCON60325.2024.10687774

[17] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., ... & Le, Q. V. (2019, October). Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (pp. 1314–1324). https://doi.org/10.1109/ICCV.2019.00140

[18] Yu, N., Davis, L., & Fritz, M. (2019, October). Attributing fake images to GANs: Learning and analyzing GAN fingerprints. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (pp. 7555–7565). https://doi.org/10.1109/ICCV.2019.00765

[19] Ramakrishnan, S. (2020). Deception in the digital age: Exploring the intersection of deepfakes and cybersecurity challenges. *International Journal of Science and Research (IJSR), 9*(9), 1611–1615. https://www.ijsr.net/getabstract.php?paperid=SR24314132532

[20] Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T., & Xie, S. (2022, June). A ConvNet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 11976–11986). https://doi.org/10.1109/CVPR52688.2022.01167

[21] Tan, M., & Le, Q. V. (2019, June). EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)* (Vol. 97, pp. 6105–6114).

[22] Li, Y., Yang, X., Sun, P., Qi, H., & Lyu, S. (2020, June). Celeb-DF: A large-scale challenging dataset for DeepFake forensics. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 3204–3213). https://doi.org/10.1109/CVPR42600.2020.00327

[23] Rössler, A., Cozzolino, D., Verdoliva, L., Riess, C., Thies, J., & Nießner, M. (2019, October). FaceForensics++: Learning to detect manipulated facial images. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (pp. 1–11). https://doi.org/10.1109/ICCV.2019.00009

[24] Dolhansky, B., Bitton, J., Pflaum, B., Lu, J., Howes, R., Wang, M., & Ferrer, C. C. (2020). The Deepfake Detection Challenge dataset. *arXiv preprint arXiv:2006.07397*.

[25] Singh, S., Sarala, P., Chandra, S. K., & Kumar, M. D. (2024). DeepFake image detection and classification using EfficientNet model. In *2024 OPJU International Technology Conference (OTCON) on Smart Computing for Innovation and Advancement in Industry 4.0* (pp. 1–5). IEEE. https://doi.org/10.1109/OTCON60325.2024.10687774

[26] Suryal, A. (2024). *RETRACTED: Leveraging metadata in social media forensic investigations: Unravelling digital clues – A survey study*. *Forensic Science International: Digital Investigation, 50*, 301798. https://doi.org/10.1016/j.fsidi.2024.301798

[27] Ticovan, I. V., & Sebestyen, G. (2025). Forensic data analysis pipeline. *Acta Universitatis Sapientiae, Informatica, 17*(1), 1–22.

[28] Solanke, A. A., & Biasiotti, M. A. (2022). Digital forensics AI: Evaluating, standardizing and optimizing digital evidence mining techniques. KI - Künstliche Intelligenz, 36(2), 143–161. https://doi.org/10.1007/s13218-021-00724-3

[29] Iyengar, S. S., Nabavirazavi, S., Hariprasad, Y., Prasad, H. B., & Mohan, C. K. (2025). Future of AI-driven digital forensics. In Artificial intelligence in practice: Theory and application for cyber security and forensics (pp. 335–364). Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-39974-3_15

[30] Tyagi, A. K., Kumari, S., & Richa. (2024). Artificial intelligence-based cyber security and digital forensics: A review. In Artificial intelligence-enabled digital twin for smart manufacturing (pp. 391–419). https://doi.org/10.1002/9781119980774.ch13