Scientific Research Publishing

Journal of Computer and Communications, 2021, 9, 120-131
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Countermeasure against Deepfake Using Steganography and Facial Detection

Kyle Corcoran, Jacob Ressler, Ye Zhu

Department of Electrical Engineering & Computer Science, Cleveland State University, Cleveland, OH, USA
Email: k.e.corcoran@vikes.csuohio.edu, j.t.ressler@vikes.csuohio.edu, y.zhu61@csuohio.edu

How to cite this paper: Corcoran, K., Ressler, J. and Zhu, Y. (2021) Countermeasure against Deepfake Using Steganography and Facial Detection. *Journal of Computer and Communications*, **9**, 120-131.
https://doi.org/10.4236/jcc.2021.99009

Received: June 26, 2021
Accepted: September 27, 2021
Published: September 30, 2021

Abstract

As deepfake technology continues to advance at a rapid pace, there is a constant need to develop new methods to counteract its use. Physical copies of authentic items like signed baseball memorabilia use a certification to identify the authenticity. This paper proposes using steganography to embed a signed watermark inside a digital image. By using RSA to generate, sign, and verify the watermark, individuals will be able to authenticate personal images or try to verify signed images for authenticity. We evaluate the proposed approach with images manipulated by deepfake algorithms. The experiment results show 100% detection rate on deepfake images. The signing time and the verification time are around 300 ms according to our experiments. So the overhead of the countermeasure are negligible.

Keywords

Deepfake, Deepfake Detection, Steganography, Cryptography, Facial Detection

1. Introduction

1.1. Background

Deepfake is an emergent and rapidly developing technology that uses machine learning to generate synthetic media that replaces a person in a video or image with someone's likeness. Deepfakes rely on neural networks that use a set of data samples to learn to mimic attributes of a person's face with training. Neural networks are non-linear models for predicting or generating content based on an input [1]. Deep fakes are usually created by using a combination of several of the following networks: Encoder-Decoder, Convolutional neural Networks, Generative Adversarial Networks. This technology was once very expensive but has

become much more accessible and powerful over the years it has existed. Malicious actors have taken advantage of this to generate believable fake media for various unethical purposes such as deception, blackmail, and mischaracterization.

1.2. Deepfake Algorithms

This paper uses an encoder and decoder algorithm to generate deepfakes. Encoder is a separate neural network that has the job of taking in dataset x of images and encoding them into a vector En(x) = e. The decoder, the other neural network, has the job of taking the vectors created by the encoder and attempting to turn this representation back into faces as closely matching the input as possible De(En(x)) = x.

In order to detect how well the neural network is doing encoding and decoding faces a loss and weight function is used. The loss function is used to update the weights with an optimization algorithm like gradient descent. A loss function for training model M as an n-class classifier, where the output of M would be the probability vector $y \in \mathbb{R}^n$. Forward propagation is applied to M to obtain $y' =$ M(x). Loss function is computed by comparing y' to the probability vector y, then back-propagation is performed to update the weights. The Loss $\mathcal{L}$ over the entire training set X is calculated as

$$\mathcal{L} = -\sum_{i=1}^{|X|}\sum_{c=1}^{n} y_i[c]\log\left(y_i'[c]\right) \tag{1}$$

where $y[c]$ is the predicted probability of x_i belonging to the c^{th} class [1]. Once the model has evaluated its performance the weights are updated for both Encoder and Decoder algorithm. This is repeated many times constantly updating the weights based on the loss values, theoretically improving the reconstruction of input face (**Figure 1**).

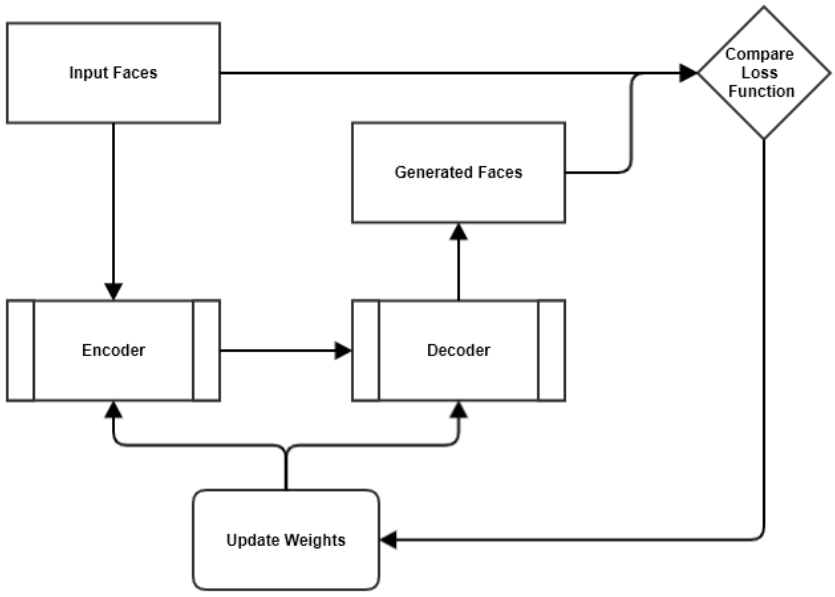

Figure 1. Encoder and decoder (figure caption).

Next the neural network uses the same encoder for both A data set (source face) and B data set (target face). This concept is called Shared Encoder and allows for the encoder to learn one algorithm to construct both faces. The idea is to tell the neural network to take the encoding of one face and decode it using the other face. When training the model two decoders are generated one for decoding face A and the other for decoding face B. To perform the deepfake faceswap the encoding of face A En(A) is passed to the decoder of face B De_B(En(A). Resulting in a swapped face is the output of the model (**Figure 2**).

1.3. Our Approach

In this paper we discuss the application of digital signing watermarks on images that are embedded using steganography. This approach uses facial detection to bound the embedded watermark inside the subject's face of the input image allowing for higher accuracy in detecting deep fake face swaps. The watermark is a fixed length and retrieved at run time with the asymmetric private key. The private key is used to sign the watermark using Rivest Shamir Adleman (RSA) decryption algorithm. The algorithm returns a string of bytes that is used to flip the least significant bits of the bitmap returned from facial detection. This allows for the signed watermark to be embedded in the face of the image giving the illusion the image was not changed to the human eye. Once the image is signed steganalysis is used to retrieve the embedded signed watermark. If the image has had a encoder and decoder swap the pixels of the face the watermark retrieved will not be verified using RSA verification. If the image has not been manipulated, then the verification will return a success authenticating the image has not been face-swapped.

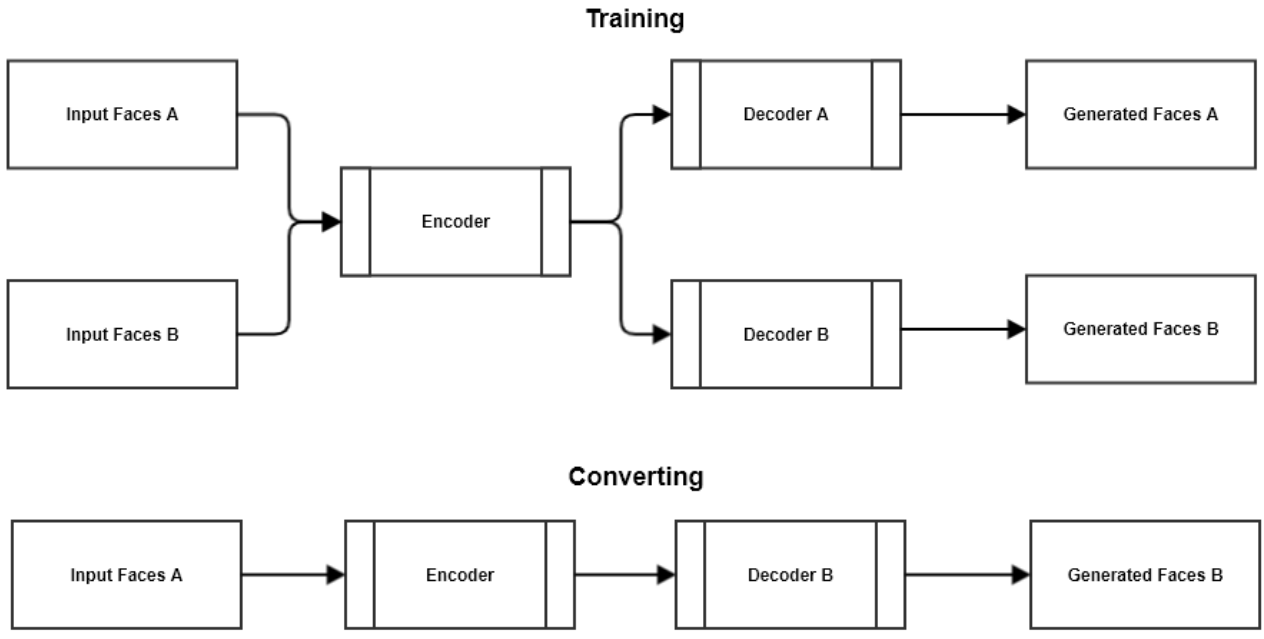

Figure 2. Shared encoder and swapped.

1.4. Our Contributions

This paper focus on using steganography, steganalysis, cryptography, and facial detection for media forensics to detect face swap deepfakes. Related work has focused on detecting deepfakes using media forensics that analyzed the unique fingerprints left behind by Generative Adversarial Networks (GAN) [2]. Our contribution is a novel approach to detection of deepfakes.

1.5. Structure of This Paper

Section 2 of this paper looks at different algorithms that are related to the model proposed. It is broken up into three parts describing the related works of deepfake, countermeasures and steganography. Section 3 overviews the model proposed and is broken into two parts. Section 3.1 is general overview of the whole model both sign.py and verify.py. Section 3.2 is a detailed view of both sign and verify. Section 4 goes over the evaluation of how well the model performed. It is comprised of three parts each detailing the evaluation of the performance. Sections 4.1 and 4.2 detail our two performance experiments. Section 4.3 discusses some identified issues using steganography with facial detection and proposes solutions. Section 5 is the conclusion and future works section. This section talks about how well the performance of the model did overall and future implementations and research.

2. Related Work

2.1. Deepfake Algorithms

This paper focuses on detecting face-swapped deepfake images. Face swapping used to be a manual process done with photoshop prior to 2004 [1]. Online communities began finding improved ways to perform face swapping with deep neural networks. The original deepfake network used an encoder and decoder neural network. Over time improvements have been made to the model configurations, including adversarial training, residual blocks, a style transfer loss and mask loss to improve the quality of face and eyes [3]. FaceSwap [4], is an open-source tool and used for deepfake face swap in this paper. Software comes with popular implementations, including deepfake lab, and multiple variations of the original deepfake network for face-swaps.

2.2. Deepfake Countermeasures

Deepfakes generate artifacts which are subtle to humans but can be easily detected using machine learning and forensic analysis [3]. Detection measures focus on these artifacts, this paper is heavily focused on the forensics of the pixels in an image. A pixel bitmap is the grid of pixels represented by bits. Each pixel of the image is represented by a gray scale or red, green, blue (RGB) pixel. These pixels are stored in the pixel bit map by their corresponding color number represented in bytes. By analyzing the pixels of an image extreme precision can be made on detecting manipulation of image. This type forensics uses a combi-

nation of cryptography and steganography to authenticate and verify images.

2.3. Steganography

Steganography is the hiding of information inside information. This paper focuses on hiding information inside images using least significant bit replacement (LSB). A portable network graphics (PNG) or a joint photographic expert group (JPEG) image is stripped down to its bitmapped image format (pixel bitmap). The (LSB) implementation places the embedded data at the least significant bit of each pixel in the bitmap. The bitmap is the reconstructed and outputs a PNG with the embedded data [5]. Steganalysis is the retrieving of data in steganography. This is done by stripping a PNG into its original bitmap and using the reverse algorithm of steganography to retrieve original data.

3. Detecting Deepfake Images

3.1. Overview of Our Detection Method

The model proposed in this paper embeds a cryptographic watermark to the 8th bit of each byte in the pixel bitmap. The bounds are determined at runtime by facial detection algorithm. This approach allows the cryptographic watermark to be embedded in the face of the image to improve accuracy of detecting face swaps. The model in this paper is implemented using two different python3.9 programs. The first program Sign.py allows the user to sign a specific image using a cryptographic watermark. The software uses TKinter as a graphical user interface (GUI) which displays successful if image's signed watermark is properly embedded. The second program Verify.py allows the user to choose a signed image and cryptographically verify it has not been manipulated. The GUI return success if cryptographic watermark is verified successfully and failed if not (**Figure 3**).

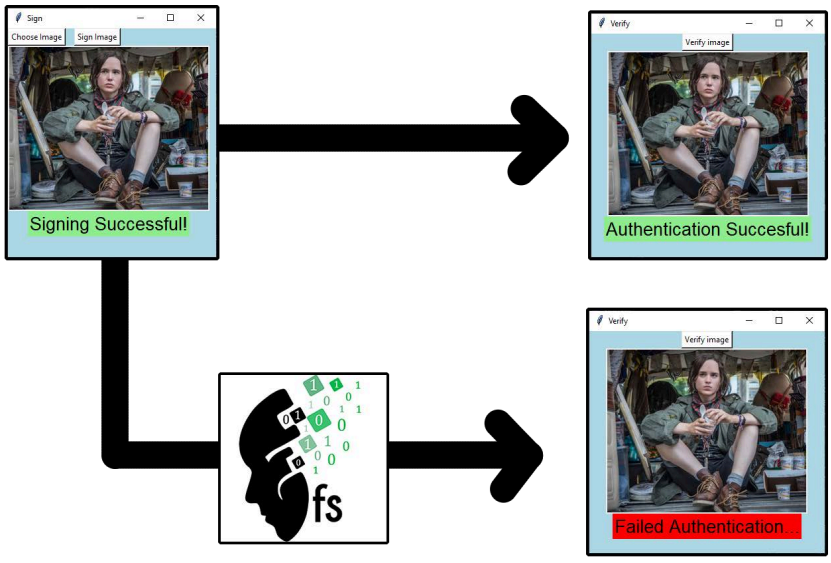

Figure 3. Model overview.

3.2. Details

The system details of Sign.py will take an input image and use a facial detection algorithm to return the bounds of the face in the image. The facial bounds will get scrapped and stored into a pixel bit map for steganography. The private key and watermark will be retrieved to use Rivest Shamir Adleman (RSA) decryption algorithm. The algorithm uses mathematical properties of prime cofactors to generate a unique private key that is used to sign the watermark where $S = D(W)$. This process returns a binary string of the signature, which is embedded inside the face, using the facial bounds bitmap, of the input image with least significant bit (LSB) steganography. The image is reconstructed from the bitmap and returned with the signed watermark embedded (**Figure 4**).

The verification software details will allow a user to input an image they wish to authenticate. Facial detection is used to return the bounds of the face in the image where the watermark is stored. The watermark can be retrieved by using the reverse LSB algorithm used to embed the watermark. Then the asymmetric public key will be retrieved to use RSA encryption algorithm to decipher the unenciphered watermark. The algorithm uses properties of trap-door one-way permutations which allow the watermark to by deciphered using the public keys encryption algorithm where W = E(S). Verified images will be deemed as successful authentication, whereas unverified images will be deemed as failed authentication (**Figure 5**).

4. Performance Evaluation

Our software was tested on two general metrics: security and usability. Security is measured through detection rate, which is whether our image verification software is able to accurately detect both real and fake images. Usability is measured through execution times for the signing and verification processes. We

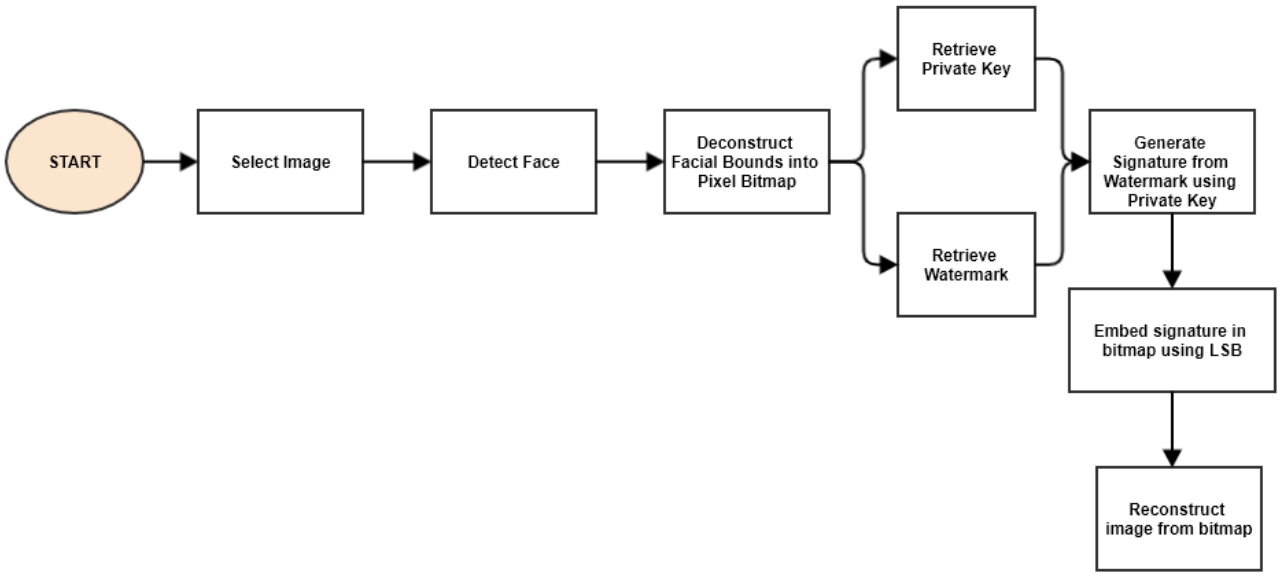

Figure 4. Sign.py model.

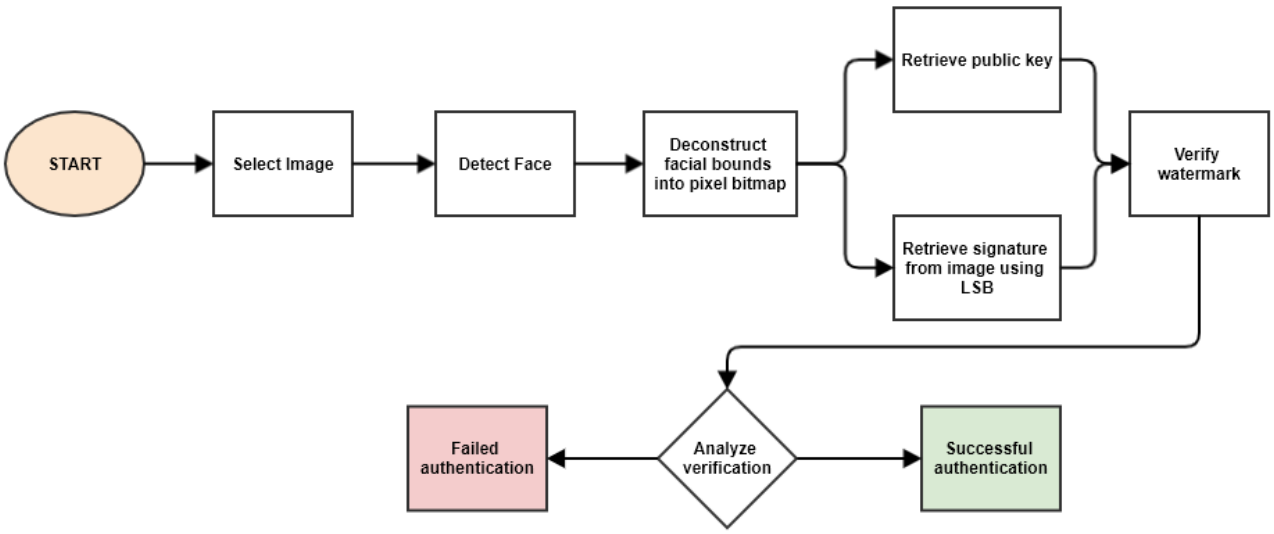

Figure 5. Verify.py model.

performed two different experiments to test these metrics, the first for general purposes and the second for testing the effects of message length on our usability metrics.

4.1. Variable Image Size Testing

In our first experiment, we performed some general testing on our software. We selected 30 different images – 10 for each of our three FaceSwap models, split evenly between the two faces trained on by each model. *Model 1* was trained on Emma Watson and Daniel Radcliffe for 400,000 training iterations using FaceSwap's *Original* trainer; *Model 2* was trained on Emma Watson and Ellen Page for 150,000 iterations using FaceSwap's *Dfl-H128* trainer; and *Model 3* was trained on Anthony Mackie and Will Smith for 60,000 iterations, once again using FaceSwap's *Dfl-H128* trainer. These variations were made to cover a general breadth of potential scenarios and subjects that could be encountered in the wild (differences in skin tone and complexity, gender, facial hair, overall quality of a deepfake, etc.).

Five pieces of information were recorded for each iteration of the experiment. *Filename* refers to the name given to each base (unsigned, real) image. *Image Size* refers to the total number of pixels in each base image. *Signing Time* refers to the time in milliseconds it takes to create a signed image from a base image. *Verification Time* refers to the time in milliseconds it takes to verify 1) the signed image and 2) the deepfake signed image. *Detection Rate* refers to whether verification correctly (indicated by 1) or incorrectly (indicated by 0) detects 1) the signed image and 2) the deepfake signed image.

The experiment procedure is as follows. We start with a base image, indicated by *Filename*. This image is then signed using our signing software, with the *Signing Time* recorded. We take this new signed image and create a deepfake of it using FaceSwap and the respective model. This is our deepfake signed image.

From there, we run both the signed image and the deepfake signed image through our verification software. The *Verification Time* and *Detection Rate* are recorded for each. The results of this experiment can be seen in "**Table 1**."

Table 1. Security and usability metrics for variable image sizes.

Filename	Image Size (px)	Signing Time (ms)	Verification Time (ms)		Detection Rate (1 or 0)	
			Signed image	Deepfake signed image	Signed image	Deepfake signed image
watson1	540,876	340	597	551	1	1
watson2	262,656	299	404	342	1	1
watson3	129,300	282	304	219	1	1
watson4	242,946	287	465	400	1	1
watson5	151,500	274	337	245	1	1
radcliffe1	138,589	287	339	254	1	1
radcliffe2	353,564	334	468	363	1	1
radcliffe3	324,000	291	414	350	1	1
radcliffe4	135,000	300	334	256	1	1
radcliffe5	89,775	289	302	280	1	1
watson2_1	5,988,000	769	2984	3003	1	1
watson2_2	8,024,000	897	4175	4624	1	1
watson2_3	635,050	348	558	489	1	1
watson2_4	3,686,400	654	2511	2492	1	1
watson2_5	2,457,600	489	1524	1547	1	1
page1	238,934	309	303	248	1	1
page2	938,002	357	627	481	1	1
page3	2,161,200	539	612	512	1	1
page4	2,160,000	501	687	572	1	1
page5	3,755,808	830	841	775	1	1
mackie1	336,000	288	324	254	1	1
mackie2	699,392	352	416	340	1	1
mackie3	472,200	322	509	448	1	1
mackie4	437,000	331	310	239	1	1
mackie5	282,534	304	367	279	1	1
smith1	315,329	294	362	260	1	1
smith2	739,000	350	452	388	1	1
smith3	834,560	349	395	314	1	1
smith4	979,900	421	430	350	1	1
smith5	261,000	286	298	215	1	1

From these results, we can see that image size directly affects both signing and verification times. Our largest image (*watson2_2* which consists of just over 8 million pixels) took 897 milliseconds to sign, 4175 milliseconds to verify as a signed image, and 4624 milliseconds to verify as a deepfake signed image. In contrast, our smallest image (*radcliffe5* which consists of just under 90,000 pixels) took 289 milliseconds to sign, 302 milliseconds to verify as a signed image, and 280 milliseconds to verify as a deepfake signed image. The main reason for this is that both our signing and verification software utilize pixel traversal in their facial detection as well as their steganographic encoding (signing software) and decoding (verification software). As such, the more pixels an image has, the longer it will likely take to perform these specific pixel-based operations on said image. Some of this could be mitigated through further optimizations to our LSB algorithm, specifically regarding decoding.

Another conclusion we can reach from our experiment results is that our software is consistently able to correctly detect both signed images and their corresponding deepfakes. The verification software had a 100% detection rate for both signed images and deepfake signed images during our experiment. That is to say, all signed images were confirmed as being authentic, and all deepfake signed images were confirmed as being inauthentic.

4.2. Variable Message Length Testing

In our second experiment, we tested the effect of different message lengths on our signing and verification times. We began with a message length of 1 byte (a single character) and doubled the length with each iteration for 25 iterations. All iterations were performed using the same image (*page1* from "**Table 1**"). Verification times recorded are only for signed images, as no deepfakes were generated for this experiment.

Three pieces of information were recorded for each iteration of the experiment. *Length* refers to the length of the message in bytes (equivalent to the number of characters in the message). *Signing Time* refers to the time in milliseconds to create a signed image from a base image. *Verification Time* refers to the time in milliseconds to verify a signed image.

The experiment procedure is as follows. We start with the base image *page1*. The base image is then signed using a signature generated from a message of the specified *Length*. The *Signing Time* of that process is recorded. The signed image is then verified using the same message as was used for signing, with the *Verification Time* recorded. This whole process is done three times for each message length, with the averages also recorded. All results for this experiment can be found in "**Table 2**".

From the results, we can see a gradual general upward trend in both signing and verification times, though we believe some further testing would need to be done using larger message lengths so as to fully grasp the effect of message length on signing and verification times. If we account for randomness, all

Table 2. Usability metrics for a constant image using variable message lengths.

Length (bytes)	Signing Time (ms)				Verification Time (ms)			
	Trial 1	Trial 2	Trial 3	Average	Trial 1	Trial 2	Trial 3	Average
1	303	303	298	301.33	268	255	280	267.67
2	286	310	282	292.67	267	257	258	260.67
4	285	305	285	291.67	270	275	256	267.00
8	281	283	290	284.67	265	259	274	266.00
16	289	282	282	284.33	258	262	273	264.33
32	283	285	286	284.67	266	264	262	264.00
64	286	286	284	285.33	278	280	274	277.33
128	289	287	291	289.00	261	265	262	262.67
256	286	276	295	285.67	260	260	252	257.33
512	280	295	296	290.33	305	256	275	278.67
1024	289	288	287	288.00	267	263	281	270.33
2048	285	297	274	285.33	267	265	257	263.00
4096	284	283	288	285.00	267	256	260	261.00
8192	290	292	302	294.67	273	260	264	265.67
16,384	303	293	297	297.67	263	262	258	261.00
32,768	304	280	295	293.00	279	256	262	265.67
65,536	314	294	300	302.67	259	264	261	261.33
131,072	300	281	282	287.67	267	261	281	269.67
262,144	278	286	282	282.00	278	262	261	267.00
524,288	275	291	306	290.67	265	270	268	267.67
1,048,576	276	298	280	284.67	283	299	278	286.67
2,097,152	285	287	278	283.33	274	261	262	265.67
4,194,304	289	289	298	292.00	279	273	275	275.67
8,388,608	298	295	291	294.67	278	280	279	279.00
16,777,216	324	316	307	315.67	306	291	302	299.67

completion times up through the 8-megabyte mark are relatively similar. The average signing times from 1 byte up to 8 megabytes have a range of 20.67 milliseconds (from 282 milliseconds to 302.67 milliseconds), and the average verification times over the same interval have a range of 21.67 milliseconds (from 257.33 milliseconds to 279 milliseconds). In both cases, however, there is a notable jump in completion time at 16 megabytes. Signing took an average of 315.67 milliseconds to complete, and verification took an average of 299.67 milliseconds, both of which are considerably greater than any prior times recorded over the course of our experiment.

We can logically assume the trend of increasingly more noticeable completion time differences would likely continue upon further doublings. However, as

stated, more testing would need to be done in order to solidify this assumption.

4.3. An Issue with Facial Detection

After our experimentation, we encountered an interesting issue with our implementation of facial detection that we feel is novel and worth discussing in some detail. Very rarely, our verification software would fail to authenticate a signed image. We found that the source of this problem was due to the application of LSB manipulation in the facial region during signing. The minor changes made to the facial pixels would sometimes be enough to slightly shift the facial bounds returned by our facial detection. This would lead to a misalignment between where the signing software encodes the signature and where the verification software checks for the encoded signature, resulting in failed authentication.

There are a few potential solutions to this problem that we have come up with. One solution is to add a start sequence to our signature before signing our image. By doing such, we would be able to use slightly broader facial bounds within our verification software to search for said start sequence. The increase in facial bounds is simply to ensure that if there is a start sequence in the image, we will be able to find it. However, this approach could still result in the failed authentications of signed images if the width of its facial bounds is not equal to that of the base image's facial bounds. If the width is less, we would miss some encoded pixels in our traversal. If the width is more, we would include unencoded pixels in our traversal. Both of these would result in incorrect signatures, and thus failed authentication.

Another solution would be to re-run our signed image through the facial detection algorithm to make sure it returns the same facial bounds as the base image. If it does not, we would create a new signature from our message and re-sign the base image. This would be done until the signed image and base image returned identical facial bounds. Such a solution could heavily impact signing times (depending on how many attempts at signing would be needed), but it would in theory guarantee the authentication of un-doctored signed images.

5. Conclusion and Future Research

In conclusion the model we have adapted is accurate when detecting constrained images. It is not perfect, however, and more research is required to identify a more sophisticated way of storing the cryptographic watermark inside the face and consistently retrieving the same bounds. Future work in this area could be done with regard to handling images with multiple faces, as well as images with no faces. Both are common scenarios to encounter that will need to be addressed in order for this model to be widely usable. Future work could also be done regarding handling video, as our model currently only supports the use of still images.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Blanz, V., *et al.* (2004) Exchanging Faces in Images. *Computer Graphics Forum*, **23**, 669-676. https://doi.org/10.1111/j.1467-8659.2004.00799.x

[2] Yu, N., *et al.* (2019) Attributing Fake Images to GANs: Learning and Analyzing GAN Fingerprints. 2019 *IEEE/CVF International Conference on Computer Vision* (*ICCV*). https://doi.org/10.1109/ICCV.2019.00765

[3] Mirsky, Y. and Lee, W. (2021) The Creation and Detection of Deepfakes. *ACM Computing Surveys*, **54**, 1-41. https://doi.org/10.1145/3425780

[4] Deepfakes. "Deepfakes/Faceswap." GitHub, github.com/deepfakes/faceswap

[5] Tiwari, N. and Madhu, S. (2010) Evaluation of Various LSB Based Methods of Image Steganography on GIF File Format. *International Journal of Computer Applications*, **6**, 1-4. https://doi.org/10.5120/1057-1378