Image and Video Authentication with Permissible Transformations

Kenneth Acquah, Ramya Durvasula, Wilbur Li, Michael Silver

May 16, 2019

Abstract

We rely on photos and videos as sources of truth in social, legal, and political contexts, so it is essential to be able to verify the authenticity of images and videos. However, transformations such as cropping and rotating directly change the content of images and thus prevent the use of simple hash-and-sign authentication. PhotoProof, a cryptographic framework developed by Assa Naveh and Eran Tromer [7], allows for images to be authenticated even after undergoing allowed transformations. A proof is produced after the image is originally taken, and each subsequent transformation verifies the previous proof and generates a new zero-knowledge proof verifying the legitimacy of the transformation. Prior to our work, the code for PhotoProof had been written only with the purpose of being used for benchmarking. To make the PhotoProof service usable, we patched the C++ core to allow for compatibility with Debian 9, packaged the Python code into a library, and built a website to allow users to apply transformations and verify image authenticity. Further, we built a GIMP plugin to enable proof generation and verification via a peer-to-peer system to remove dependence on a trusted server. In addition to implementation, we theoretically extended the PhotoProof paradigm to include transformations for blackouts, blurring, resizing, and zooming. We also provide a description of how PhotoProof can be extended for use in video and define several gadgets for video transformations.

Contents

1 Introduction

In today's world, where anyone can publish anything, it is becoming increasingly important to be able to authenticate photos and videos. With the rise of deepfakes and other adversarial image manipulation with potential political and legal implications, authentication and content integrity have become a matter of national security [4].

The problem with traditional authentication, using the hash-and-sign paradigm, is that it requires that the message is never modified. However, if after taking a photo, the owner of the photo would like to modify the image to make it better fit for publication, such as by cropping or rotating, the bitwise representation of the image will be modified. In their 2016 paper titled *PhotoProof: Cryptographic Image Authentication for Any Set of Permissible Transformations*, Assa Naveh and Eran Tromer provide a potential solution to this problem [7]. They define a framework that can be used to authenticate images which may have undergone a specific set of user-defined transformations.

PhotoProof uses the Proof-Carrying Data paradigm (PCD). Each image is defined as an N x N pixel grid, each of which has three 8-bit values to represent its color in RGB format. Additionally, each image has metadata which contains its actual height and width (N is the maximum allocated size), the authors, and any other desired information. Every image is associated with a proof which shows that it was generated from a series of valid transformations. The proof does not reveal anything about the nature of previous transformations (if any), which is important because some transformations such as cropping are used to protect anonymity.

The PhotoProof algorithm has three parts. First, the generator is used to apply the transformation. Then, the prover generates a proof based on the transformation, and the verifier is used to verify that the proof passes. Each of these functions communicates with libsnark's PCD modules and with PhotoProof gadgets which were created for each transformation.

Mathematically, the zksnarks framework is what allows for zero-knowledge proofs and transformations. Computations are based in an elliptic curve field, so that operations such as addition and multiplication can be done efficiently. Each transformation is represented as a series of quadratic arithmetic programs (QAP). For example, for the crop operation, the QAPs would check that each input pixel is equal to each output pixel for the non-cropped region and each output pixel is equal to 0 for the cropped region. In order to ensure that the transformations can be computed using operations applicable in an elliptic field, transformations and transformation-checking conditions are represented in a language called R1CS, or Rank-1 Constrain System, which represents equations as connections in an algebraic circuit.

Once the gadget-specific transformation has been defined as a QAP, it is converted to a single equation equating two polynomial products. This equation can be verified by querying the polynomials at specific locations without revealing any information about the underlying polynomial, therefore enabling zero-knowledge proofs.

While it is clear that PhotoProof provides novel functionality and has immediate use cases, the existing implementation of PhotoProof was constructed primarily as a proof of concept, and thus does not have an accompanying interface to allow it to be used by the general public. In order to make PhotoProof accessible, we patched the C++ core to make the existing codebase compatible with Debian 9 and built a Python library to wrap PhotoProof functionality. This library is accessed by a website we built to allow for transformations to be generated and verified online. Additionally, we built a GIMP plugin which allows users to directly generate transformed images and removes dependence on a trusted server. We describe our implementations in the sections below.

In addition to working on the implementation of PhotoProof, we describe theoretical extensions to the framework. First, we discuss additional gadgets which can be added to the PhotoProof implementation. Then, we explore the use of PhotoProof for video authentication. Existing methods for video authentication do not allow for any transformations, and typically rely on hashes or watermarking algorithms which reduce video quality. PhotoProof's framework, while not encompassing audio, could be extended to apply to videos. We describe a framework and several possible transformations for the video use case.

2 Implementation

2.1 Improvements to the PhotoProof Core

The PhotoProof core is a C++ library that generates proofs for certain permissable tranformations [7]. It uses libsnark to perform zkSNARKs operations and uses the GNU Multiple Precision Arithmetic Library to perform its arithmetic operations [6, 1]. In addition, the PhotoProof core includes a small Python wrapper that uses the ecdsa library to sign an image, uses the PIL library to perform the transformation, and calls the PhotoProof C++ library to generate the proof [9, 3].

When Eran Tromer first gave us access to the PhotoProof core, largely untouched since the publication of the PhotoProof paper, the code would not compile on most modern systems. We patched the C++ core to allow for compatibility with Debian 9 (with g++ 6.3 on x86-64). Additionally, we worked to clean up the Python code and allow it to work as a library for use in the PhotoProof Service (Section 2.2) and GIMP plugin (Section 2.3).

2.2 PhotoProof Service

2.2.1 System Description

We built a website that makes PhotoProof accessible and practical for everyday users. The website has two main purposes. First, it lets users apply permissible transformations to images in order to generate proofs (which we call the "generate" step). Second, it allows user to upload a transformed image along with a proof to verify whether the image is authentic (which we call the "prove" step).

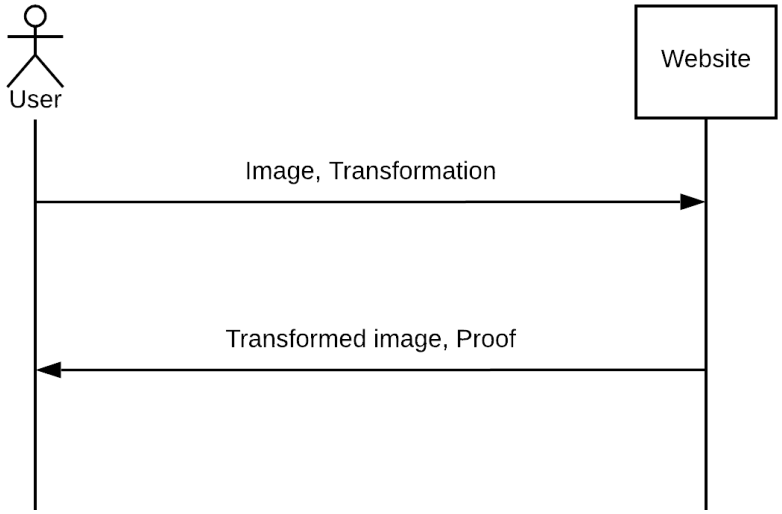

Figure 1: Flow diagram for generating proofs with an image and a tranformation. Given an uploaded image and a selected transformation, the website returns a zip with the transformed image and its proof.

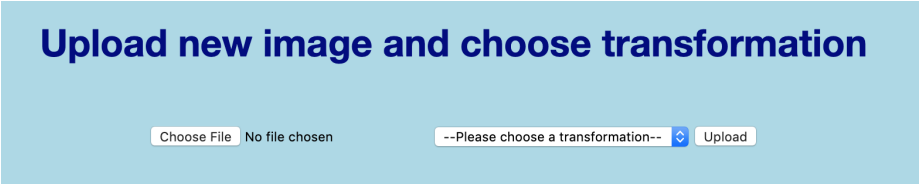

Figure 2: Screenshot of the generate page of the PhotoProof service. Given an uploaded image and a selected transformation, the website returns a zip with the transformed image and its proof.

In the generate step, the user uploads a photo and select a permissable transformation they would like to perform on the image, which the website will perform for them serverside and return the transformed image and proof back to the user as a zipfile. Figure 1 shows this workflow and Figure 2 shows a screenshot of this part of the site.

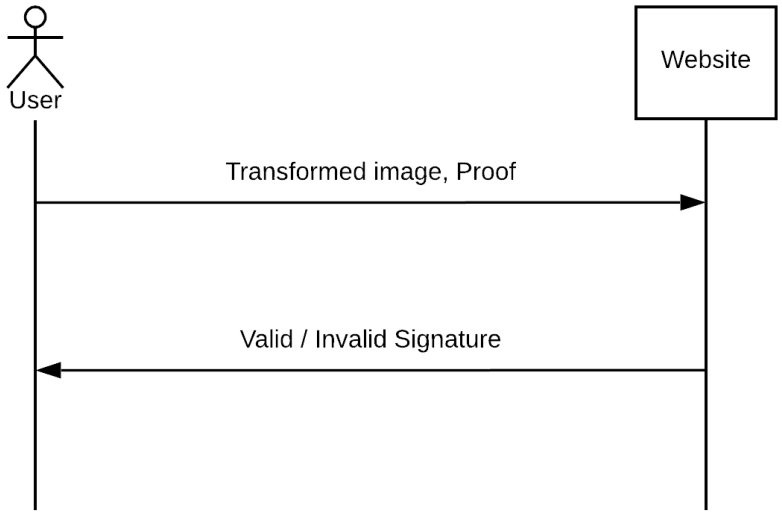

Figure 3: Flow diagram for proving a tranformed image with its proof. Given an uploaded transformed image and its proof, the website returns whether the proof is valid for the transformed image.

Figure 4: Screenshot of the prove page of the PhotoProof service. Given an uploaded transformed image and its proof, the website returns whether the proof is valid for the transformed image.

In the prove step, the user uploads the transformed image and the proof, which the the website uses to check serverside whether the proof is valid for the

image and inform the user. Figure 3 shows this workflow and Figure 4 shows a screenshot of this part of the site.

The PhotoProof service was written in Flask, a Python library for building websites [8]. The Flask server directly calls the Python wrapper of the PhotoProof Core (see Section 2.1) for both the generate and prove steps. We store images in bmp format, since that is the original format supported by the PhotoProof core, and use Pickle to serialize the proofs (originally stored as Python dictionaries) [2].

2.2.2 Evaluation

Using a PhotoProof service like the one we have built requires trust in the service and the communication with that service. If we were to launch this site for the public, this could be partially achieved by open sourcing the code and setting up HTTPS with a strong certificate. Still, the user might not trust the server (because after all, who knows what code the server is *actually* running). For the extra skeptical and security minded users, we recommend signing and verifying client side, such as with our GIMP Plugin (see Section 2.3).

Furthermore, our website has two additional security flaws. First, we disabled checks of image timestamp. The original PhotoProof code will consider the the timestamp of when the image was generated, which proves problematic since bmp files do not include a timestamp [10]. To circumvent this issue, we completely ignore the timestamp by setting the timestamp to zero unix time in both the generate and prove steps. This may make the PhotoProof service vulnerable to replay attacks; an attack could occur when an adversary submits a previously computed transformation and proof, and a verifier evaluates the transformation and proof as having been signed appropriately by the user who holds the public and secret key which were used to sign the original transformation.

Finally, the PhotoProof service does not currently support any kind of key exchange. The server uses its own keys in both the generate and prove steps, which means that if a user generated a proof on their own outside of the website with their own keys, the website would not be able to check the proof for them. Likewise, if a user used the website to generate a proof, they would not be able to check it on their own. We hope this might be implemented in the future by proving user accounts and a way for users to manage keys through the site.

2.3 GIMP Plugin

Requiring an external service to compute transformations and their respective proofs engenders particular security barriers towards widespread adoption of out application. A PhotoProof service necessitates a set of trusted servers, with which clients can interact with in order to achieve valid proofs and verifications. Alternatively, if users could compute proofs and transformations locally and send those keys over HTTPS to trusted peers, PhotoProof could be used without requiring a centralized server; this would further accomplish our project goals

of enabling any individual to prove, transform, and sign image taken with a trusted device's private key.

2.3.1 Proof Generation

To allow individuals to easily compute proofs and transformations on their personal machines, we leveraged PhotoProof's technology to construct a Python library that enables users to construct Image authentication proofs in R1CS. To transform an image and prove she has not made an illegal transformation with another agent, one must agree with the agent upon public, signing, verification keys, and compliance predicate (the set of permissable transformations) for communication (the security of PhotoProof secure constant generation is outside of the scope of this project). Once both parties have agreed upon these parameters, they can generate image authenticator objects which can prove and verify the transformations agreed upon within the compliance predicate. In addition to the Python library, we built a GNU Image Manipulation Program (GIMP) software plugin to provide a user interface with which users can construct image transformations. Once the image has been transformed, the generated proof is saved to disk and can be used to authenticate the photo created in GIMP.

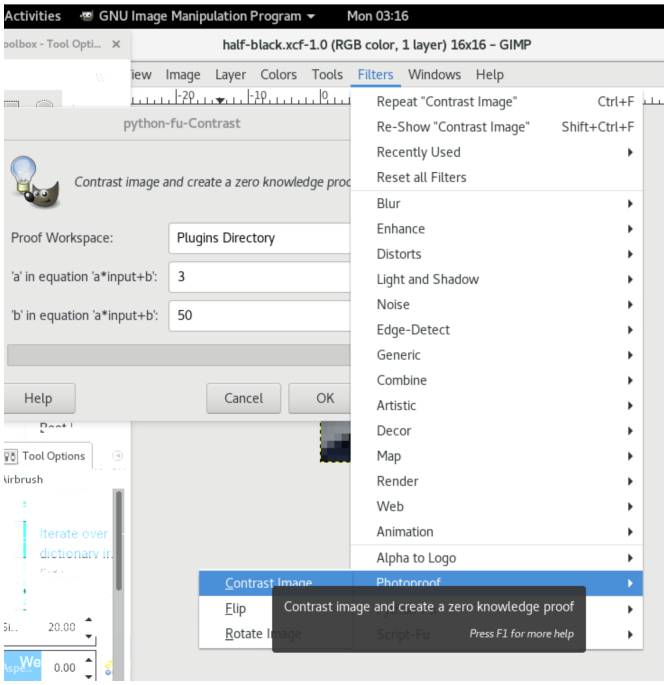

Figure 5: User interface for generate transformations and creating proofs.

2.3.2 Proof Verification

Once an individual has created a proof and transformation he can send it to other agents over an HTTPS/SSL connection. The verifier than can use their own image authentication object defined within their PhotoProof package to verify the proof and transformation, the procedure for verifying using the PhotoProof library is displayed below:

```
from photoproof import *
import pickle

ia = ImageAuthenticator(False)
with open("proof-of-image-from-alice.pickle", "r") as f:
    proof = pickle.load("proof-of-image-from-alice.pickle")
image_to_prove = Image("image-from-alice.bmp")
print(ia.verify(image_to_proof,proof))
```

Similarly, an agent could use the PhotoProof service specified in Section 2.2 to access a trusted server that can verify others transformations and proofs.

2.3.3 Evaluation

The our software implementation of PhotoProof presents nontrivial security limitations. With a primary security goal of ensuring that all permissiable transformations evaluate to true, and all nonpermissable transformations evaluate to false our implementation faces the following obstacles.

- GIMP Plugin software uses allows GIMP to call remote procedures specified as "plugins" in GIMP, While executing a transformation, our plugin is run concurrently within the GIMP version a user is interfacing with. It would therefore be possible for a corrupted GIMP executable corrupt proofs as they are created, forcing them to evaluate to false. In addition, if verification were also implemented a GIMP plugin, a corrupted GIMP executable could relay false positive verification results to an unknowing user, comprimising the security of our workflow.
- Communication pitfalls similar to those mentioned in 2.2.2.

2.4 Limitations

The largest limitation of PhotoProof from the implementation standpoint is its speed. PhotoProof Core (Section 2.1) is incredibly inefficient. For a 16×16 image Naveh and Tromer measured it takes an average of 16.92 seconds to generate a proof and 13.97 seconds to verify the proof. We measured it takes on the order of minutes to generate a proof and seconds to verify. For this reason, both the PhotoProof Service (Section 2.2) and the GIMP plugin (Section 2.3) use 16×16 images, which still take several minutes each to generate proofs. Making PhotoProof faster is an essential future improvement to make it a practical system for wide adoption.

3 Additional Image Gadgets

At present, PhotoProof contains implementations for six gadgets: flipping, transposition, rotation, cropping, brightness/contrast adjustment, and the identity. Although these gadgets allow for a number of useful transformations, they are not sufficient for most of the real-world use cases. We describe four new gadgets—blackout boxes, pixelation, and resizing—which, if implemented would significantly enhance the capabilities of PhotoProof.

Blackout boxes and pixelation are two operations which would be applied to a rectangular region within the image to protect anonymity or sensitive material. For example, they may be useful for witnesses disclosing sensitive imagery or video footage to reporters, where some but not all of the image content can be shared. Resizing and zooming are both simple image operations which allow for content publishers to incorporate images in their media in an accessible format. While these are not the only transformations with common use cases, they can be combined with the original six transformations to allow a wide variety of transofrmations.

When defining these gadgets, we continue to use the R1CS scheme from [5], which is built into the libsnark library. R1CS allows us to move from a Algebraic Circuit to Quadratic Arithmetic Program (QAP). QAPs enable the use of Linear PCP and then zkSNARK.

3.1 Blackout Boxes

We define a new blackout gadget which allows us to black out a rectangular region. Its parameters are x_1, y_1, x_2, y_2, the coordinates of the lower left and upper right corners of the region to be blacked-out. The algorithmic circuit functions almost like the crop gadget, such that each pixel of input is sent through a MUX gate and mapped to each pixel of the output. Output pixels within the rectangular region will receive the value 0 for each of the R, G, B colors; all other pixels will receive their original values. Metadata must be changed to indicate that blackouts are present, so that blackout boxes are not used to produce a false impression inside the image.

3.2 Pixelation

We similarly define a pixelation operation which pixelates a rectangular region. It takes as parameters x_1, y_1, x_2, y_2, the lower left and upper right corners of the rectangle, and a parameter a representing the side length of the blurring grid squares. For pixels which are not within the rectangular region, the input pixels are mapped directly to output pixels. Within the rectangular region, the pixels are partitioned into $a \times a$ grid squares, and then all of the a^2 pixels in a square are given the average pixel value of that square. This effect can be visualized in Figure 6.

If we bound the size of the pixel squares to be less than or equal to A, the complexity of this gadget varies with $O(A^2 * N^2)$. We can pass these $a \times a$

pixel squares into an average computation, and then fill the output square with that average value. Real-valued arithmetic is done using the fixed-point representation used for rotation and brightness/contrast adjustment in the original PhotoProof paper.

If our rectangle cannot be broken down perfectly into these $a \times a$ squares, we fill the rest with partial grids that together form the required region, which can also be seen in Figure 6.

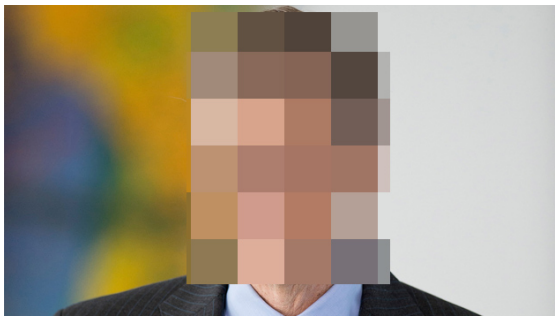

Figure 6: Pixelated image showing how we can have grid squares that take on the color of their pixels' average value.

3.3 Resizing

Pictures are rarely taken in the correct resolution for their end use. We recognize this and enable downward resizing, which yields a downsampled copy of the original image. Similarly, we want to enable resizing images to larger than the original size, up to a limit of $N \times N$ since we are constrained by the pure size of our circuits. Input parameters here are W, L, which are the target width and length for the length and width of the output image.

This is potentially a very expensive operation requiring a circuit up to $O(N^4)$ in size if performed naively, since different resizing operations require completely different connections between input and output.

We can improve on this by setting up an overall resize as two resizings applied in parallel, one length-resize followed by one width-resize. In this configuration, each pixel will only need to be connected to N others in the same row or the same column. This results in an overall resizing process that consists of two N^3 processes, for an $O(N^3)$ overall complexity.

We can enhance the quality of each 1D resize by mapping interpolations of old pixels into new pixels. If a new pixel is to be made two old pixels x, y, then the new value $t = \alpha x + \beta y$, where α, β are determined by where the old pixels fall. This achieves the same result as using the bilinear interpolation algorithm for resizing.

4 PhotoProof for Video

PhotoProof utilizes PCDs to enable the property of ensuring authenticity of a transformed image without revealing information about the original untransformed image. We understand the value of this service and look to expand it to videos.

As a starting point, we model a video simply as a succession of T frames that are each $N \times N$ pixels, with no special encodings or embeddings would be practical in the real world. This model does not take into account audio.

We first describe how the main PhotoProof algorithm would need to be adjusted to adopt video authentication and then compile a list of video gadgets we would like to design.

4.1 Video transformations

We list the transformations we seek to cover for videos and images. Some transformations are applied to individual frames or are meant to apply differently to different frames. We refer to these as *frame transformations*.

Transformations like trimming that inherently affect multiple frames will be referred to as *sequence transformations*.

4.2 PCD-enabled video authentication

We rely on PhotoProof's technique to authenticate videos. A description of the original implementation can be found at [7].

Algorithm 1: Compliance Predicate $\Pi^{\tau}(z_{in}, z_{out}, t, \gamma) \rightarrow 0/1$. This checks if $z_{in} \rightarrow z_{out}$ is a valid transformation t with the parameters γ. We ultimately use this predicate to verify the validity of incoming images.

Algorithm 2: Generator $G_{PP}(1^N, 1^{\lambda}) \rightarrow (pk_{PCD}||p_S, vk_{PCD}||p_S, s_S)$. Here our proving key $pk_{PP} = pk_{PCD}||p_S$, and signing key $sk_{PP} = vk_{PCD}||s_S$ come from a concatenation of the Proof-Carrying Data scheme and a standard set of public/secret keys.

Algorithm 3: Prover $P_{PP}(pk_{PP}, E_{in}, \pi_{in}, t, \gamma) \rightarrow (E_{out}, \pi_{out})$. Here the video E is edited according to the transformation t and parameters γ. Proofs are converted into PCD form, and then our final output is the transformed video and a PCD proof.

Algorithm 4: Verifier $V_{PP}(vk_{PP}, E, \pi) \rightarrow 0/1$. The verification key vk_{PP} can now be parsed as $vk_{PCD}||p_S$. Depending on what π is, we return $V_S(p_S, I, \pi)$ or $V_{PCD}(vk_{PCD}, (I, p_S), \pi)$ which is a true/false bit.

4.3 Video gadgets

In order for videos to be properly transformed under our PCD approach with trust in the final video but no knowledge of the original video, we must produce circuit gadgets that output constraints in Rank 1 Constrained Systems (R1CS) for these specific transformations.

PhotoProof constructs gadgets considering them as circuits in order to enable conversion from equations into R1CS form, so we continue to do that here.

4.3.1 Frame gadgets

Existing image gadgets. Flip, transpose, contrast, brightness, rotate, and crop were all defined in the original PhotoProof paper, and we can keep these circuits the same.
Zooming. Frame zooming would be important for a transformation that adjusts the zoom inward, a common video editing technique that should not alter the authenticity of the rendering. A fluid approach would traverse the original image, remove the excess borders around the desired portion. It would then perform a resize operation to change this cut version to the original size. The computational complexity of a zoom gadget would be similar to that of a crop and resize, and would probably naively be around $O(T_L \cdot N^3)$ in size, which would be large for a zoom operation lasting T_L frames.

4.3.2 Sequence gadgets

Continuous gadgets facilitate transformations that collectively affect many frames or the metadata of the video itself.
Trimming Trimming is an essential operation for video manipulation. In order to maintain the security goals, we only allow for trimming at the ends of a video, but prevent users from deleting part of the middle of a video. This tranformation requires t_s, t_e, a start time and an end time, and will map all pixels in all frames outside of this segment to 0. The metadata then needs to be updated to reflect the new length of the video.
Frame rate The frame rate is set in the metadata of the video. Frame rate should be adjusted as a macro change for the entirety of the video, since changing the frame rate of individual snippets can harm the authenticity of the video itself due to inconsistency. The metadata will contain the required frame rate, and should also indicate whether the rate has changed since the original value.

5 Acknowledgements

We would like to thank Professor Ronald L. Rivest, Professor Yael Tauman Kalai, and the 6.857 course staff for advising us and supporting us with this project. We would also like to thank Eran Tromer and Assa Naveh for giving us access to the C++ and Python code for PhotoProof and answering our questions.

References

[1] Gnu multiple precision arithmetic library. https://gmplib.org/.

[2] Pickle: Python object serialization. https://docs.python.org/2/library/pickle.html.

[3] Pillow: The friendly pil fork. https://python-pillow.org/.

[4] Wait, is that video real? the race against deepfakes and dangers of manipulated recordings. https://www.usatoday.com/story/tech/2019/05/13/deepfakes-why-your-instagram-photos-video-could-be-vulnerable/3344536002/.

[5] E. Tromer E. Ben-Sasson, A. Chiesa and M. Virza. Scalable zero knowledge via cycles of elliptic curves. *Advances in Cryptology– CRYPTO 2014.*, 2014.

[6] SCIPR Lab. libsnark: a c++ library for zksnark proofs. https://github.com/scipr-lab/libsnark.

[7] Assa Naveh and Eran Tromer. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 255–271. IEEE, 2016.

[8] Armin Ronacher. Flask: The python micro framework for building web applications. http://flask.pocoo.org/.

[9] Brian Warner. Python ecdsa. https://github.com/warner/python-ecdsa.

[10] Wikipedia. Bmp file format. https://en.wikipedia.org/wiki/BMP_file_format.