PhotoProof in Golang

David Mberingabo
Carnegie Mellon University

1 Introduction

This project is an implementation of the PhotoProof cryptographic scheme in Go. I first introduce the scheme and its underlying cryptographic guarantees, then provide a threat model for a specific use case and explain the steps I took to implement some of the features involved in the scheme.

PhotoProof, as presented by Naveh et al., is a cryptographic scheme for image authentication. [7] Using a trusted camera and PhotoProof, a photographer can digitally sign an original photograph and define a list of *permissible transformations* that can be used to edit the image in the future. The signed photograph and the set of permissible transformations are used to create a proving mechanism. This mechanism allows any *editor* to prove that only permissible transformations, as defined by the photographer, were applied to the original photograph. Editors can derive a proof that the editing was indeed permissible and share this proof with the public. This proof can then be used by *verifiers* to verify an image and determine if the image was legitimately derived from an original image and only underwent editing that is authorized by the photographer.

Modern digital editing tools allow people to create fake photographs that appear realistic. Photo-journalists, legal courts and businesses often need to authenticate an photographic evidence as real or fake. Naveh et al. define *Image Authentication* as "Image Authentication (IA) is, loosely speaking, the ability to prove that an image faithfully represents some original photograph that was captured in a given class of physical image acquisition devices (e.g., a camera model)." Without image authentication, propaganda, scams and falsified legal evidence become prevalent.

In practice, some alterations to an image do not hinder the "authenticity" of the image, so it is important to capture authenticity even in cases where the image may have been permissibly edited. Examples for edits that do not cause an image to lose its authenticity are rotations or lossy compression. This makes digitally signing an image a poor authentication mechanism because any rotation would make the image inauthentic under such a simple scheme. PhotoProof attempts to create an Image Authentication mechanism that supports some set of permissible transformations.

PhotoProof uses zero-knowledge Succinct Arguments of Knowledge (zk-SNARK) cryptographic proofs to create an Image Authentication scheme. A zk-SNARK proof leaks no information about the argument being proven, other than the proof's validity. [2] For example, one could check the authenticity of an image by verifying the validity of its proof, without knowing or learning anything about the set of permissible transformations or any previous edits the image underwent. This allows PhotoProof to maintain privacy for images that were edited, such as cropping out a section of an image without revealing there was any cropping at all. With zero-knowledge proofs, the editor does not have to reveal the cropped out section in order to prove their resulting image was permissible under the photographer's rules. Of course, the photographer can choose to reveal the list of permissible transformations if they wish to show that no cropping is permissible; they can do this by including the list of transformations inside the image's metadata upon initially signing it. Compared to its predecessor Image Authentication schemes, PhotoProof achieves a negligible error probability in authenticating a fake image or rejecting a real image. Beyond this, PhotoProof is also provably secure against computationally bounded adversaries that might try to forge convincing images and pass them as authentic.

2 Background

A zk-SNARK is a cryptographic primitive that is used to enforce properties of computations that are executed by distributed, untrusted parties. For example, in a multi-party computational environment a node A may receive an input from a previous node B, and node A must verify the input is indeed the result of a computation that was executed according to publicly agreed-upon computational properties. Once verified, node A may conduct a computation on the input, and it may include its own secret local parameters, before passing the

computation's output to a node C along with a proof that the output is indeed the result of a chain of verifiable executions. This is accomplished without revealing those local secret parameters to node C. This means that node C does not need to know any other node's secret parameters or any previous inputs to verify that the output from A is valid under publicly agreed-upon computational properties.

Each node along the multi-party computation runs a verification and proving process using a *compliance predicate*, which is the set of computational properties each execution across the nodes must adhere to. If a node's execution is compliant, then they are able to create a proof for their execution's output and participate as a legitimate node in the distributed computation. A distributed computation can be thought of as a graph, where each node verifies its input, runs a compliant execution with local secret parameters, and creates a proof of compliance for its output. Using zk-SNARKs as proofs, the final output and its proof are all one needs to verify the final output was computed by compliant nodes in the graph. This means the speed of verification, given a final output and its proof, is independent from the size of the distributed computation graph. Since only the final output is required, the proof's size is also independent of the size of the distributed computation graph. This makes zk-SNARKs efficient proof schemes for maintaining compliance accross multiple untrusted execution nodes with secret local parameters.

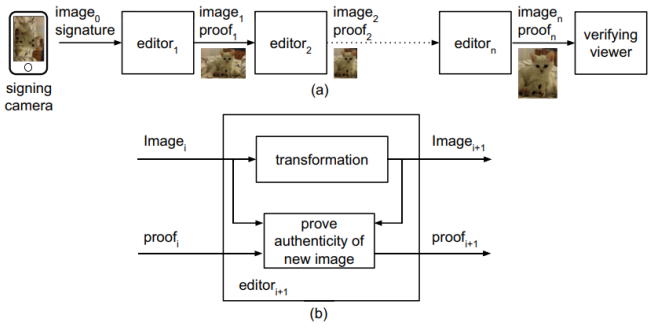

Figure 1: The history of an image as a distributed computation (a) where in each step a PCD-generated proof is appended (b). [7]

2.1 How are zk-SNARKs used in PhotoProof

A zk-SNARK consists of three algorithms: a Generator (G_{PCD}), Prover (P_{PCD}) and Verifier (V_{PCD}). The Generator algorithm generates the prover and verifier keys (pk_{PCD}, vk_{PCD}) from a public *compliance predicate* (Π) and security parameter (λ). Any prover and verifier can use their respective keys to run the Prover and Verifier algorithms on public and secret inputs.

In PhotoProof, the Prover algorithm requires (pk_{PCD}), an input message, a corresponding proof ($\pi_i n$), some local parameters and an output message ($msg_o ut$). With these parameters the Prover algorithm outputs a new proof ($\pi_o ut$) that its output message adhered to Π. The Verifier algorithm uses vk_{PCD}, the new proof $\pi_o ut$ and the output message $msg_o ut$ to succinctly determine whether the computational graph that created $msg_o ut$ was indeed compliant to Π.

The zk-SNARK proofs used in PhotoProof are difficult to forge for computationally bounded adversaries and ensure that the only information gained, given an image and proof, is the authenticity of the image. PhotoProof uses zk-SNARKs to succinctly verify the authenticity of an image. The work process begins with an administrator deciding on a set of permissible transformations and the photographer taking and signing a photograph with a trusted camera. Then the administrator runs the PhotoProof's generator algorithm to generate a proving key and a verification key. Editors are given proving keys and can generate proofs that an image they transformed is still within permissible bounds. So the edited image can still be verified by any individual with the verification key, the proof and the image.

In PhotoProof, an image I is an NxN matrix and some metadata M (i.e. $I = (NxN, M)$). The matrix contains integers that make up pixels, with each integer ranging from 0 to 255. Given an image I, we say I has *permissible provenance* in image J, where I is the result of a series of permissible transformation on J. J is not necessarily the original image that was first taken and signed by a trusted camera. We say an image I is authentic if is has permissible provenance from an original image O, with respect to the set of pre-defined permissible transformations (T), the signed original image (σ) and its signature verification key (p_s).

Image Authentication (IA) in PhotoProof is described as $IA(T) = (S, G_I A, P_I A, V_I A)$, which translates to "an Image Authentication process, with respect to a set of permissible transformations T, requires an unforgeable signature scheme S, a Generator algorithm $G_I A$, a proving algorithm $P_I A$ and a verification algorithm $V_I A$". The Generator algorithm takes a image size N and a security parameter λ to generate a secret signing key $sk_I A$, a public verification key $vk_I A$ (this is a tuple containing a verification key for both the zk-SNARK proof and the digital signature) and a proving key $pk_I A$ for generating proofs. $G_I A$ is a pre-processing step and assumed to be executed only once and in advance by a trusted party. The Prover algorithm takes an input image ($I_i n$) and its proof ($\pi_i n$), a transformation (t) and its parameters (γ), and a proving key $pk_I A$, and it outputs a transformed image $I_o ut$ and its proof $\pi_o ut$. The Verifier algorithm simply takes an image, its proof and a verification key and returns whether an image is authentic or not.

The set of permissible transformations and the digital signature of the image are used to create a compliance predicate, which must be adhered to by all the nodes in the system, where

Algorithm 1 compliance predicate $\Pi^{\mathcal{T}}(z_{in}, z_{out}, t, \gamma)$

Input: incoming and outgoing messages $z_{in} = (I_{in}, p_{in})$, $z_{out} = (I_{out}, p_{out})$, an image transformation t and a parameter string γ.
Output: 0/1.

1: **if** $z_{in} = \perp, t = \perp$ **and** γ is a digital signature **then**
2: **return** $V_{\mathsf{S}}(p_{out}, I_{out}, \gamma)$
3: **if** $t \in \mathcal{T}$ **and** $t(I_{in}, \gamma) = I_{out}$ **and** $p_{in} = p_{out}$ **then**
4: **return** 1
5: **return** 0

Figure 2: This is Π, the PhotoProof compliance predicate, as defined by Naveh et al. Where *p* is a proof and *I* is an image. This is implemented in my code demonstration. [7]

nodes are editors in PhotoProof. This compliance predicate is expressed as a arithmetic circuit, in a computer language, and translated into a constraint system using a zk-SNARK construction such as Groth16. [5]

Algorithm 2 `PhotoProof` generator $G_{\mathsf{PP}}(1^N, 1^\lambda)$

Input: a maximal image size N and a security parameter λ.
Output: a proving key pk_{PP}, a verification key vk_{PP} and a signing key sk_{PP}.

1: $(s_{\mathsf{S}}, p_{\mathsf{S}}) \leftarrow G_{\mathsf{S}}(1^\lambda)$
{generate a secret key and a public key of the signature scheme}
2: generate an $\mathbb{F}_p$-R1CS instance C_N which computes $\Pi^{\mathcal{T}}$ when applied on N-images.
3: $(pk_{\mathsf{PCD}}, vk_{\mathsf{PCD}}) \leftarrow G_{\mathsf{PCD}}(C_N, 1^\lambda)$
{generate PCD keys}
4: **return** $(pk_{\mathsf{PCD}}||p_{\mathsf{S}}, vk_{\mathsf{PCD}}||p_{\mathsf{S}}, s_{\mathsf{S}})$

Figure 3: This is the Generator algorithm for PhotoProof. This is also implemented in my code demonstration, as described by Naveh et al. [7]

3 Threat Modeling

3.1 Use Case: PhotoProof against Rendered Images

Computer generated images, like photoshopped images, are created by a computer program. Nowadays specific instructions can easily be given as input to machine learning models that were trained on real images to generate photo-realistic renderings according to the given instructions. AI-enhanced CGIs can be derived from a photograph of a real scene making photo-evidence manipulation extremely easy.

Algorithm 3 `PhotoProof` prover $P_{\mathsf{PP}}(pk_{\mathsf{PP}}, I_{in}, \pi_{in}, t, \gamma)$

Input: a proving key pk_{PP}, an N-image I_{in}, a proof π_{in}, an image transformation t and a parameter string γ.
Output: an edited image I_{out} and a proof π_{out}.

1: parse pk_{PP} as $pk_{\mathsf{PCD}}||p_{\mathsf{S}}$
2: **if** π_{in} is a digital signature string **then**
3: $\pi'_{in} \leftarrow P_{\mathsf{PCD}}(pk_{\mathsf{PCD}}, \perp, \perp, \pi_{in}, (I_{in}, p_{\mathsf{S}}))$
{"convert" the signature to PCD proof by calling the PCD prover}
4: **else**
5: $\pi'_{in} \leftarrow \pi_{in}$
6: $I_{out} \leftarrow t(I_{in}, \gamma)$
7: $l \leftarrow (t, \gamma)$
8: $z_{in} \leftarrow (I_{in}, p_{\mathsf{S}})$
9: $z_{out} \leftarrow (I_{out}, p_{\mathsf{S}})$
10: $\pi_{out} \leftarrow P_{\mathsf{PCD}}(pk_{\mathsf{PCD}}, z_{in}, \pi'_{in}, l, z_{out})$
11: **return** π_{out}

Figure 4: This is the Prover algorithm for PhotoProof. This is also implemented in my code demonstration, as described by Naveh et al. [7]

Algorithm 4 `PhotoProof` verifier $V_{\mathsf{PP}}(vk_{\mathsf{PP}}, I, \pi)$

Input: a verification key vk_{PP}, an N-image I and a proof π.
Output: 0/1.

1: parse vk_{PP} as $vk_{\mathsf{PCD}}||p_{\mathsf{S}}$
2: **if** π is a digital signature **then**
3: **return** $V_{\mathsf{S}}(p_{\mathsf{S}}, I, \pi)$
4: **if** π is a PCD proof **then**
5: **return** $V_{\mathsf{PCD}}(vk_{\mathsf{PCD}}, (I, p_{\mathsf{S}}), \pi)$
6: **return** 0

Figure 5: This is the Verifier algorithm for PhotoProof. vk_pp is a tuple that contains both the PCD verification key (vk_PCD) and the signature's public key (p_s). This is also implemented in my code demonstration, as described by Naveh et al. [7]

CGI can be used to create realistic or stylized images and animations for a variety of applications, including movies, video games, and product visualizations. However, CGI can also be used to create propaganda, impersonate individuals and create bot accounts to bypass user verification mechanisms. This has created a societal need for methods of discerning the difference between CGI and real scenes photographed with a camera. Authentication-by-picture is used by many modern applications nowadays to determine bot from human.

Images are often used in court and other trust-based proceedings that are fundamental to societal stability, therefor realistic image renderings become useful for malicious actors. In such trust-based proceedings, a digital photograph that represents an actual physical scene and underwent authorized transformations should be discernible from an image rendered by a computer.

Using PhotoProof, image transformations can be defined as permissible or not by an administrator. However, there remains the question of whether the photographer (owner of the camera) truly took the image with a trusted camera. After all, the photographer could have rendered the image and created a proof for it using their camera's private key and there would be no way to determine if the image came from the camera or not. PhotoProof solves the question of provenance from a trusted camera, but it does not necessarily solve the question "was this image rendered or not?"; the question it does answer is "did this image come from an trusted camera or not?"

It is important to realize that the PhotoProof scheme, as solely a zk-SNARK implementation, makes no explicit distinctions between creating a proof for a rendered image or an image from the camera. In the paper's "Additional features and extensions" section, Naveh et al. suggests a certificate system where manufacturers issue private keys to cameras and publish public key certificates, which can be used to attest that an image has its provenance from one of the manufacturer's cameras. This can be incorporated into PhotoProof by proving that a image was taken with a trusted camera, given the manufacturer's public key certificate as a parameter to proofs and the secure embedding of the signing key inside a trusted camera.

For better trust and security, these certificates can be stored on a secure and decentralized ledger. A public key represents a photographer's camera and it should remain unmodified, otherwise the wrong owner could be associated to an image they never took.

Beyond that, photographers may want to prove their images are not rendered by proving the images can not be altered inappropriately. A compliance predicate for proof against rendering can be agreed-upon by the public to include permissible transformations that do not dramatically alter the image. This compliance predicate could be securely embedded in the trusted camera and the camera could output a proof using the compliance predicate against renderings, and include the permissible transformations as part of the public parameters of the proof. This requires trusting a party to create a proof, using the compliance predicate against renderings, in a secure and trusted manner. I do not describe or implement this compliance predicate against renderings. It goes to show that PhotoProof is not infallible and must be implemented according to its specific use case, in order to offer stricter proofs against renderings.

3.2 Key Theft, Image Injection and 2D Scene Staging

With PhotoProof, Naveh et al. proposed improvements to security limitations that arise when the photographer attempts to bypass the camera's security features. Malicious photographers can break the camera's temper-proofing and cause side-channel attacks that would allow them to sign a rendered image with the camera's embedded private key or inject a fake sensory input to be signed as an authentic image. Malicious photographers would have an advantage since they own the camera and have ample time to attempt various attacks. Once a side-channel attack is found, it can be made easy to use and widely spread, which can be devastating for the camera manufacturers as their authentication guarantees would quickly become meaningless.

Naveh et al. propose a revocation mechanism in the case these private keys become breached. This is a remedy to key breach, but this does not avoid the actual hardware vulnerabilities: side-channel and fault injection attacks, reverse engineering. The issue of securing the physical camera from a malicious photographer's side-channel attacks is beyond the scope of this project and the PhotoProof paper, however, it is a key part of any trusted camera.

In their Appendix, Naveh et al. describe specific remedies to attacks like *2D Scene Staging* and *Image Injection*. 2D scene staging is when a malicious photographer holds up a 2D scene in front of the camera to trick the camera that it is a real 3D scene. Image injection on the other hand is when an attacker with physical access to the camera conducts a side channel attack involving sensory data injection. These are incredibly difficult to avoid and require approaches like securing the link between image sensors and Image Signal Processors, and depth recognition.

A *3D Scene Staging* will always be deemed real by PhotoProof, no matter how secure or trusted a camera and image authentication scheme is.

4 Technical Approach

Naveh at al.'s implementation of PhotoProof was in C++. They used the open-sourced library *libsnark* to design zk-SNARK circuits and demonstrate PhotoProof. [7] In my implementation, I demonstrate PhotoProof in Golang and use the *gnark* library, which is open-source and similar to libsnark.

Go is meant to be a user-friendly language and gnark was developed by ConsenSys, a $3.2 billion software foundry, to help developers in constructing schemes based on zk-SNARK. Although I could not find any deployed projects utilizing the gnark library or experimental research, it boasts higher performance than other libraries like libsnark.

Both Libsnark and Gnark have optimized implementations of elliptic curves used for proving schemes. The *gnark-crypto* library provides curves and cryptographic primitives (like the MiMC hash function) that have been optimized for building zero-knowledge proof systems. [3] My implementation uses the optimized BN254 curve; which is used in Elliptic Curve Digital Signature Algorithms. [2] PhotoProof uses a NIST 192p elliptic curve, however, gnark does not offer any such curves, instead, the Baby JubJub companion curves are offered as BN254 in gnark. [1]

Groth16 was first introduced by Jens Groth in 2016 as a specific implementation of zk-SNARKs. [5] In Groth16, compliance predicates are expressed as part of the arithmetic circuit that represents the computation being verified. It is implemented by gnark which offers a user-friendly frontend API for writing arithmetic circuits that represent compliance predicates, under the hood. Groth16 is used in decentralized projects like ZCash and Filecoin [6].

PhotoProof defines each image transformation (Rotation, Flip, Transpose, Cropping, Contrast/Brightness), and includes an *Identity* transformation which does nothing to an image. Permissible transformations can be written as an arithmetic circuit. The circuit and its constraints are expressed in the 'Define()' function that runs on a *circuit* struct. Circuit structs contain secret and public fields which can be a single 'frontend.Variable' (interpreted as a big int) or an array of 'frontend.Variable'. Permissible transformations can be expressed through gnark's 'frontend.API', which contains arithmetic operations such as assert equality, addition, multiplication, division, etc..., that can run on the big int fields in a circuit. [4] The arithmetic circuit expressing the transformation and digital signature verification of the original image is then translated into a compliance predicate that can be used to generate prover keys – which are used to create proof that only permissible transformations were ran on an authentic image– and verifier keys – which are used by verifiers to verify if an image is authentic or not.

An administrator can select which transformations, in the form of arithmetic circuits, should be allowed to be executed on a signed image. If some other transformation, that is not an instance of a permissible transformation, is executed on an authentic image, the Prover algorithm will not be able to create a valid proof and it will be exposed as inauthentic by the Verifier algorithm. I write the Generator, Prover and Verifier algorithms for Image Authentication as described in the Naveh et al. paper. This workflow is demonstrated in the programming part of the project, but only for the Identity transformation due to limitations discussed below.

4.1 Project Limitations

In the demo, I simulate taking a picture and signing the JSON encoded N by N image (including its metadata) with a trusted camera. The trusted camera also generates a proving and verifying key. Using the verification key, the image and a PCD proof (or a digital signature), I demonstrate verification of an image. I also demonstrate how a prover would edit and create a new image, but only for the Identity transformation. The other transformations require 2D array manipulation, unfortunately, gnark has yet to implement 2D array variables in their frontend API.

My demonstration ends with an Identity transformation and taking the project further would require a library update from gnark. Due to time and skill, I was unable to manipulate 2D arrays in gnark, although I tried reaching out to gnark's GitHub issues page. With 2D array manipulation implemented, gnark would be able to handle all transformations described in PhotoProof. Gnark is a fairly new project and is not in stable version yet, so we can expect further features being implemented. I demonstrate this 2D array limitation in the demonstration and describe it further within the code attachment to this paper.

This project narrowly focuses on implementing some of the PhotoProof functionalities in Golang. To accomplish this, we first yield to these limitations, as our system does not provide practical ways of removing them:

1. Manufacturer can see the camera's private key.
2. Manufacturer can add backdoors in the camera.
3. Photographer can bypass the camera's hardware security.

References

[1] Jordi Baylina Barry WhiteHat and Marta Bell´es. Baby jubjub elliptic curve. *Ethereum Foundation, Iden3 and Universitat Pompeu Fabra (https://iden3-docs.readthedocs.io/en/latest/iden3$_r$epos/research/publications/zkp*
standards − workshop − 2/baby − jubjub/baby − jubjub.html), 2019.

[2] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 781–796, USA, 2014. USENIX Association.

[3] Gnark by Consensys. Prove schemes and curves. *(https://docs.gnark.consensys.net/Concepts/schemes$_c$urves)*.

[4] Consensys. gnark/frontend package. *Golang package description*.

[5] Jens Groth. On the size of pairing-based non-interactive arguments. 2016.

[6] Eric Lin. Gnark, concepts, prove schemes and curves. retrieved from: https://docs.gnark.consensys.net/concepts. 2023.

[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, 2016.