**[Slide 1] xx**

Good afternoon, ladies and gentlemen. It’s an honor to be standing here today and thank you to the organizers of ACDF for allowing me this opportunity.

Please use the QR code or the GitHub link on the screen to access the public GitHub repo for this presentation. The GitHub repo contains the project’s codebase, this presentation and transcript, as well as some relevant academic papers.

My name is David Mberingabo. I am the Cybersecurity Lead at RMSoft, where I focus on Secure Development Lifecycle for Next.JS web applications. I have my Masters in Privacy Engineering from Carnegie Mellon University, in Pittsburgh and my Bachelor’s in both Computer Science and Finance from Northeastern University, in Boston.  
  
  
**[Slide 2] – Content - xx**

My objective today is to give you a quick rundown of a project I’ve been working on in my spare time. It’s called PhotoGnark.

To understand this project, I will start by describing the new cutting-edge cryptographic technology called Zero Knowledge proof (also known as zk-SNARKs), the different steps involved in implementing cryptographic schemes that use Zero Knowledge proofs, and what areas are we seeing actual usage of these proofs. Once we have a concept of what Zero Knowledge proof are, then we will explore a package called Gnark which allows us to create schemes based on Zero-Knowledge proofs, and finally I will describe how that package can be used to achieve Image Authentication for Any Set of Permissible Transformations, with the PhotoProof scheme.  
  
It sounds super technical, but believe me when I say that “if I can understand it and build with it, so can you.”

I have tried to make this presentation as simple as possible and have avoided the super technical parts. This presentation should give you a good basic understanding of Zero-Knowledge Proofs and how they can be used to solve some of our modern privacy problems.

**[Slide 3 & 4] -- Zk-SNARK – Requirements -- xx**  
  
Raise your hand if you have never heard of the term zk-SNARKs or Zero-Knowledge Proofs? [interact with audience]

Zk-SNARK is an acronym for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge. Zero Knowledge Proofs are a cryptographic primitive, like encryption, hashing or digital signatures.

Zk-SNARKs involve 3 types of participants: the Administrator, the Prover and the Verifier.  
  
In brief terms, Zero Knowledge Proofs let a prover convince a verifier that they know some local secret input to a computation (a program, circuit, or polynomial equation) without revealing that secret input. The computation in question is defined by the Administrator.

[Next]

Let’s explore the acronym a little closer to better understand this new cryptographic primitive.

* **Zero-Knowledge** means that some of the local input data remains secret when the prover creating a proof of knowledge.
* **Succinct** simply means that the verification process for zk-SNARK proofs is fast for the verifier.
* **Non-interactive** means that the verifier only needs one proof to verify a chain of computations, instead of multiple interactions between the Prover and Verifier.
* And finally, **Argument of Knowledge** is a term that is a little more complicated, but we will define this term in later slides.

zk-SNARKs let a prover convince a verifier that they know a secret input to a computation without revealing it, and the acronym describes the qualities involved.

Don’t worry if this still sounds confusing, we will explore these concepts multiple times over the course of this presentation. But for now, I want you to remember the participants involved in zk-SNARK schemes: The Administrator, The Prover and the Verifier.  
  
The Administrator defines the computation, the prover defines the secret inputs and create a proof of knowledge, and the verifier verifies the proof.

**[Slide 5] -- Zk-SNARK – Where is it Used? - xx**

Zero-Knowledge Proofs (aka zk-SNARKs) were introduced in 2010, and they are now popular and widely adopted on large systems, including Ethereum and other privacy-focused chains like Zcash, as well as emerging applications in gaming, banking and Finance. Even Binance has recently implemented zk-SNARKs to improve efficiency and privacy.

These organizations pay bounties to anyone who can break their zk-SNARK implementation.   
  
zk-SNARKs are now a battle-tested cryptographic primitive, just like RSA or SHA256, with over $3.5 billion in value relying on zk-SNARKs today.

[if still under 7 mins]  
  
For example, Starknet leverages zk-SNARKs for on-chain gaming, where micro-transactions can occur without a bank being involved. Zk-SNARKs also allow players to prove certain game moves, without revealing the details of the game move, such as proving that a card was randomly drawn from a deck, without revealing to the whole server exactly which card was drawn.

**[Slide 6] -- Zk-SNARK – Arithmetic Circuits - x**

Zk-SNARK stands for Zero-Knowledge, Succinct, Non-Interactive, Argument of Knowledge, and we defined all these properties except for what is an Argument of Knowledge.

To answer the question “what is an Argument of Knowledge”, we must first familiarize ourselves with what an arithmetic circuit is.

An arithmetic circuit is simply some computation, let’s call it ***C***, that takes inputs from a finite field **F**, and returns a single output, also from the finite field **F**.

A finite field is simply a set of numbers from 0 to p-1, where p is a prime number greater than 2.

This graph represents an arithmetic circuit.

Starting at the bottom of the graph on the screen, we see the circuit takes **x1** and **x2** as input variables, as well as the constant 1. These inputs are on a finite field **F**, and undergo a series of operations (addition, subtraction, multiplication) and this result in the output **x**.

As you can see above the graph, an arithmetic circuit is used to essentially describe a polynomial function or any other computation. In other words, if you follow those arrows from the inputs, through the operators and to the final output x, you end up with that polynomial function.

**[Slide 7] -- Zk-SNARK – Example of an Arithmetic Circuit - x**

Consider a circuit used to hash a message “***m***”***.***

This circuit checks that some given hash digest ***h*** is equal to the hash of a message ***m***.

The circuit must run a hash function on ***m*** and then assert that the resulting hash subtracted from the given hash digest ***h*** equals 0. Thereby asserting that the hashes are equal.

If the given hash equals the hash calculated by the circuit, then the circuit returns 0, otherwise it returns some non-zero result. This is an example of a circuit that checks hash equality.

**[Slide 8] -- Zk-SNARK – Argument of Knowledge - x**

Argument of Knowledge is a zk-SNARK property, and with our new understanding of arithmetic circuits, we can start defining what is an Argument of Knowledge.   
  
An arithmetic circuit can be used to implement what is called an Argument System.   
  
Argument systems involve an Administrator, a Prover, a Verifier and a circuit ***C***.

In this example, x & w are local inputs to a circuit, where ***x*** is a public input value (also known as a “public statement”), and ***w*** is a secret input (also known as a secret witness).  
  
By creating a **Proof of Knowledge** and sharing it, the Prover is arguing to the Verifier that they have knowledge of both ***x*** and ***w***, such that the circuit, in this case the circuit ***C(x,w) = 0***. To create a proof, the prover needs a Proving Key, which is generated by the Administrator and shared with all participants.  
  
The Prover will NOT be able to create a Proof of Knowledge that convinces the Verifier, unless the Prover knows a ***w*** and ***x***, such that ***C(x,w) = 0***, and has access to the Administrator’s Proving Key.

The shared proof does not reveal the secret input ***w*** and the Verifier remains only aware of the public statement ***x***. Using the Verifying Key, which is also provided by the Administrator, the Verifier is able to verify the Proof of Knowledge and confirm that the Prover indeed knows a ***w***, such that ***C(x,w) = 0.***

**[Slide 9] -- Zk-SNARK – Requirements (Revisited) - xx**

zk-SNARKs of course involve much more math than we can discuss today. It typically involves encoding circuits and values to achieve efficient mathematics over “elliptic curves” and rigorously proving the properties we described. But if you want further details, please read the paper found at the QR code on the screen or by googling the title “**Proof-Carrying Data and Hearsay Arguments from Signature Cards”**. Now, with our knowledge of circuits and argument systems, let’s revisit the zk-SNARK requirements.

1. Zk-SNARKs are zero-knowledge because the proof itself does not reveal anything about the secret input ***w***, commonly called a secret witness.
2. Zk-SNARKs are succinct because the proofs are fast to verify, and they are non-interactive because only one proof is necessary for the Verifier to be convinced.
3. Zk-SNARKs are Proofs of Knowledge used in Argument Systems that involve a Prover and a Verifier, where the Prover argues they know a compliant witness ***w***, and a malicious Prover is unable to make a convincing argument.
4. Zk-SNARKs achieve completeness only if the Verifier accepts proofs of knowledge that complies to the Administrator’s circuit.

**[Slide 10 & 11] -- Zk-SNARK – General Steps Involved x**

All zk-SNARK implementations typically follow these steps:

* Step 1 -- Administrator defines a circuit, also known as a compliance predicate, with input variables that can be secret or public. The compliance predicate is known to the public, but the variables are either public or secret. The Admin then generates Proving and Verifying keys, which are to be shared with all participating nodes.

[next]

* Step 2 -- Prover nodes use the Proving key, and some local data as secret and public input, to create a Proof of Knowledge of the secret input. The Prover then shares both the proof and the public inputs to the next node. Sometimes the public inputs is public information available to anybody.
* Step 3 -- Verifier nodes use the Verifying Key, the proof and a public statement to check that the Prover knows the secret values that fulfil the Administrator’s circuit (also known as a compliance predicate). A compliance predicate is essentially a circuit that can be proven and verified.

Zk-SNARK implementations are usually thought of as multi-party computation scenarios with the help of a network graph composed of nodes. At each hop, the proof is a claim of compliance to the Administrator’s circuit and knowledge of secret input data. This is all done with the use of the Proving and Verifying keys, which all nodes have access to and is provided by the Administrator, during the first step.

**[Slide 12] -- Gnark – What is it? - xx**

Gnark is an open source, fast zk-SNARK package for the language Go. The Gnark package offers a high-level API to design circuits. It was developed by Consensys, which is a famous company that builds open-source tools used by Ethereum developers. Consensys was founded by a co-founder of Ethereum and has developed world-famous apps, such as MetaMask.

Gnark is trusted by organizations like Zcash, Linea and WorldCoin with a combined market capitalization of $4 Billion.

**[Slide 16 - 21] -- Gnark – Hashing Example - x**

[Quick explanation of Golang objects, fields & functions.]   
  
In the Gnark package, circuits are objects (or structs in Golang [point to Hashing\_Test\_Circuit]), and the circuit’s inputs are the objects’ fields. In Gnark these fields must either secret or public. In our example the Img field is the secret input and the Hash field is the public input. This means the Prover can share the Hash but not the actual image, and they want to prove that the Image they know is equivalent to the public Hash.

**[Slide 17] -- Gnark – Hashing Example - x**

Functions can only manipulate frontend.Variable types, which is why Gnark-friendly versions of objects only have this type as a field.   
  
On the right of the screen, we have an image object with normal Gnark types such as []byte, and the Pixel object is also made up of integers. On the left, we have the Gnark-friendly version of an Image object, called Fr\_Image.   
  
Both the normal Golang image and the Fr\_Image can be hashed using a normal hash function or a Gnark-friendly hash function.

Gnark provides two hashing functions, one for Gnark-friendly types and another hashing function for normal Golang types.

**[Slide 18] -- Gnark – Hashing Example - x**

Like all zk-SNARK implementations, the first step is to define a circuit, which we just did, and now the Administrator can compile our circuit into a compliance predicate and generate the proving and verifying keys, which we can then share with the Prover and Verifier nodes.

[Next]

The Prover, when given proving keys and access to the circuit definition, can use local input data to generate a proof.

We start by creating an image and its hash, which will be used as input to the Prover’s circuit instance. Then this prover’s circuit, alongside the proving key, can be used to generate a proof, as long as the inputs are valid.

[Next]

The Verifier can then use the public inputs and dummy secret inputs to recreate a circuit, which can then be used to verify a proof, with the help of a Verifying Key.

**[Slide 19] --** **Gnark – Limitations - x**

Gnark offers no security against side-channel attacks, but that’s the case with all Golang applications, this is because the Go language has no constant time guarantee. This means that we must assume a trustworthy system that is not compromised, like a secure computer.

**[Slide 20] -- Images in Modern Society – xx**

Now that we understand a little about zk-SNARKs and how they can be implemented with Gnark, let’s talk about some more useful cases than just verifying hashes are equal. Let’s talk about Image Authentication with the help of Zero-Knowledge Proofs.

Why is it so important to have Image Authentication?

The modern world relies on images in personal, commercial, legal and security contexts, as well as forming public opinion through photojournalism and social media.

Without image authentication we risk an increase in propaganda, fraud, evidence falsification in courts, extortion, and we risk a decrease in our ability to assess intelligence sources, which leads to social mistrust and an overall lack of social cohesion.

**[Slide 21 - 23] -- Image Forgery & Authentication – xx**

As image forgery becomes cheaper over time, incentives to flood the internet with fake images increases.  
  
The problem is that only 58% of fake images can be detected by non-professional human inspectors, while professional image analysts are expensive to train and higher, and have a low work capacity.

In other words, image forgery is getting better and cheaper, and our capability to detect forgeries remains slow, expensive and inaccurate.

[Next]

It’s now normal to see comments like “Wow AI images are getting scary good”, or people on X asking “Grok, is this image real??”.   
  
Here’s an example of a NASA image posted on X. In the comments below it, someone authenticates the image by sharing the NASA website link. Indeed you could reverse image search to find who may have posted the image, but what if NASA itself were to post a forgery?

In my opinion, image verification should be as simple as a browser extension. Users should be able to right-click on an image and select the “Verify image” button, or select an “Edit & prove image” button that would only allow for permissible transformations and create a proof for the output image.

[Next]

As an example, this fake image of Katy Perry at the Met Gala that went viral on X.com. The post raked in over 16 million views, with many commenters expressing their shock that the photo had been made using AI.   
  
This is not be a big deal but Imagine what kind of social damage adversaries can now do with AI image editing tools and AI bots being able to manipulate narratives.

**[Slide 24 & 25] – PhotoProof – What is it? -- x**

PhotoProof is what my project was based on, but my own project is named PhotoGnark, because it uses the Gnark library.

So let’s talk about PhotoProof: it’s a paper presented in 2016, at the IEEE Symposium on Security and Privacy, by Assa Naveh & Eran Tromer; professors at the Blavatnik School of Computer Science, Tel Aviv University. PhotoProof was recently recreated by Ke-Han Li et. al as a collaboration between the National Taiwan University, Washington State Universiy and the University of Alabany. All to say that this PhotoProof stands on strong academic credentials.   
  
In fact, Eran Tromer, one of the authors of PhotoProof was one of the academics that initially introduced ZK-SNARKs back in 2010.

[next]

PhotoProof leverages ZK-SNARKs to prove that an image originates from a Secure Camera and that it has only undergone permissible transformations,   
  
A Secure Camera is a tamperproof camera with protection against Image Injection attacks.

A permissible transformation is an image transformation that does not change an image’s authenticity. These permissible transformations are selected by the owner of the Secure Camera, also called the Administrator in this case. The Administrator defines a compliance predicate that allows participating nodes to prove an image’s authenticity, by checking that only permissible transformations occurred on the image. For example, if we crop someone’s face out of an image to protect their privacy, the image may still be considered authentic under some circumstances, but it’s up to the owner of the Secure Camera (the Administrator) to define what is a permissible transformation. So cropping 10% of the image or increasing contrast by 10% may be permissible, if the Administrator allows it.

[Next]

With PhotoProof, editors with a proving key would be able to create a proof of knowledge and compliance. This proof would claim that an output image is the result of a permissible transformation and the original image comes from a secure camera. This proof verification would be possible without revealing what image transformations actually occurred. For example, the values of cropped-out pixels would not need to be revealed to a Verifier. All the Verifier would need is the output image, a proof and a verification key from the owner of the Secure Camera.

**[Slide 26 & 27] – PhotoProof - Compared to Other Solutions – xx**

Let’s go through the steps of this zk-SNARK scheme.

The starts with the Administrator defining what are Permissible Transformations by writing a Gnark circuit, they would generate the proving and verifying keys and share them with all participating nodes. In this example the signing camera is owned by the Administrator.

In this example, the prover is the editor, who initially receives the original image with a proof that the Administrator signed the image with a secure camera, then the Prover provides a public and secret inputs to the image transformation circuit, which generates a valid proof if the inputs adhere to the compliance predicate.

[next]

Most solutions have some or all these limitations:

* Fixed set of permissible transformations,
* Non-negligible error probability,
* Vulnerability to adaptive attackers,
* Lack of succinct verification step,

The PhotoProof authors reviewed other solutions compared to PhotoProof and If you look at the bottom of the table you can see how PhotoProof scores against these other solutions. It provides a larger set of permissible transformations, a negligible error probability, and the fastest verification step.

**[Slide 28] – Limitations – x**

PhotoProof, and even zk-SNARK packages themselves, need more development for it to become useable and widely adopted.  
  
Secure systems, like secure CPUs and secure light sensor units and secure Image Signal Possessors are still undergoing serious development to mitigate attacks that leverage hardware vulnerabilities.