Using Zero to Attack Zero-Knowledge Proof (ZKP) PLONK

Nguyen Thoi Minh Quan

How to say my name?

- My name “Quan” is a prefix of **quan**tum 🤣

Goals

- ❑ To have **non**-zero “knowledge” about zero-knowledge proof (ZKP) 😁
- ❑ To have fun with number 0

Serious advice 😁

- ❑ Everything in modern ZKP is related to polynomials.
- ❑ **Learn polynomials!!!**

Agenda

- ❑ Background: introduce **intuition** of ideas, techniques and terminologies in ZKP
- ❑ How theory guides the attack's direction?
- ❑ Why does the attack work in practice?

Warm up

- Find integers x, y, z ⩾ 0 such that

$$x^n + y^n = z^n, n > 2, n \in N$$

Warm up

- Find integers x, y, z ⩾ 0 such that

$$x^n + y^n = z^n, n > 2$$

- x = y = z = **0**

Warm up

- Given “unknown” random number r, find x such that

$$r * x = 0$$

Warm up

- Given "unknown" random number r, find x such that

  $$r * x = 0$$

- x = **0**

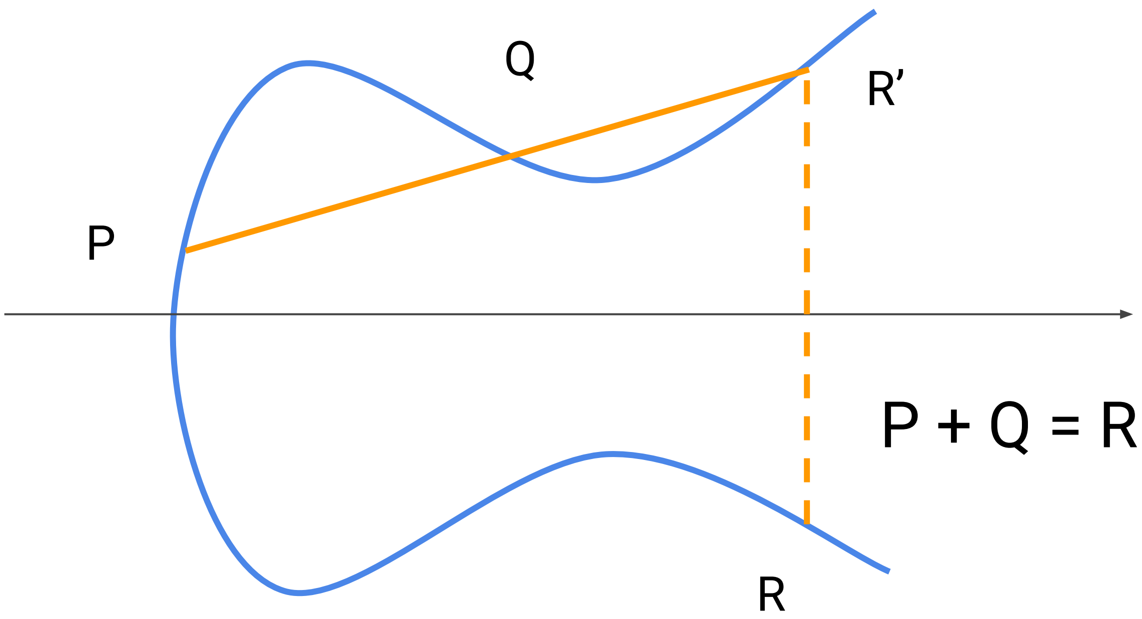
Warm up

- Using Fermat's little theorem: given p prime and x, find $x^{-1} \bmod p$
- $x^{p-1} = 1 \bmod p \rightarrow x^{p-2} * x = 1 \bmod p \rightarrow x^{p-2} = x^{-1} \bmod p$
- What will happen when $x = 0$? $x^{-1} = x^{p-2} = 0^{p-2} = 0 \bmod p$ → Inverse of $0 \bmod p$ is 0 ?!?

Why 0?

- To bypass zero knowledge/signature verification, the attacker has to find x, y, … such that f(x, y, …) = 0

Elliptic Curve



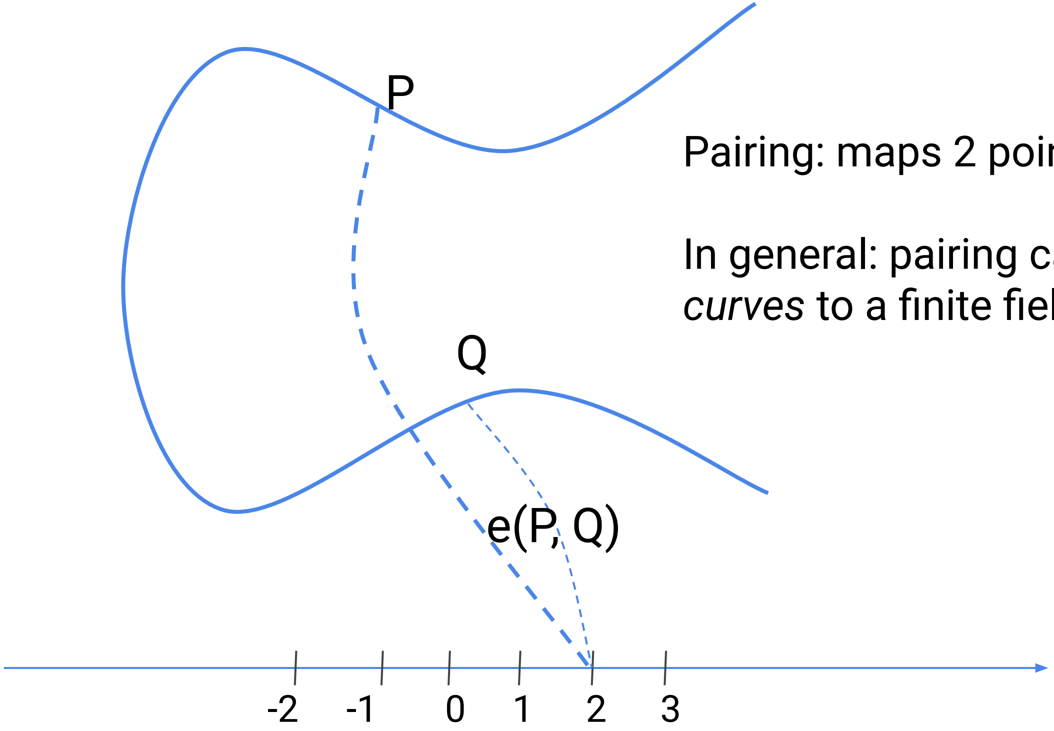
Elliptic Curve Group Structure

- Addition: P + Q
- Zero (infinity) point: P + 0 = 0 + P = P
- qG = G + G + … + G = 0, q is the order of the point.
- Group (0, G, 2G, …, (q - 1)G)

0 in Elliptic Curve

$x.0 = 0, \forall x$

Pairing



Pairing: maps 2 points to a finite field

In general: pairing can maps 2 points in *2 curves* to a finite field.

Pairing

- ❑ $e(P + Q, R) = e(P, R) * e(Q, R)$
- ❑ $e(aP, bQ) = e(P, Q)^{ab}$
- ❑ $e(aP, bQ) = e(P, Q)^{ab} = e(abP, Q) = e(bP, aQ)$: pairing helps *checking multiplication* relation

Pairing with 0

$$e(0, X) = 1 = e(Y, 0),\ \forall X, Y$$

Basic ZKP protocol and terminologies

Interactive Schnorr protocol

Prover has private key w, public key $W = wG$.
The prover wants to convince the verifier that it "*knows*" the private key w *without revealing any information* about w.

Prover

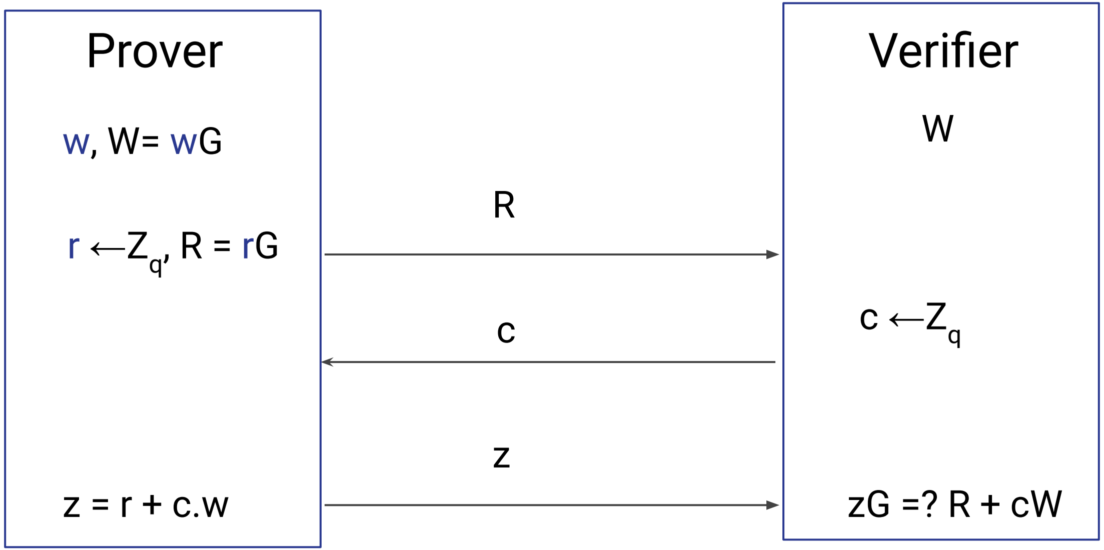
w, $W = wG$

Verifier

W

Interactive Schnorr protocol

Prover wants to convince the verifier that it "*knows*" the private key w *without revealing any information* about w



Note: zG = (r + c.w) G = rG + c.w.G = R + cW

This protocol is the basis of digital signature

Why does verifier know nothing about w?

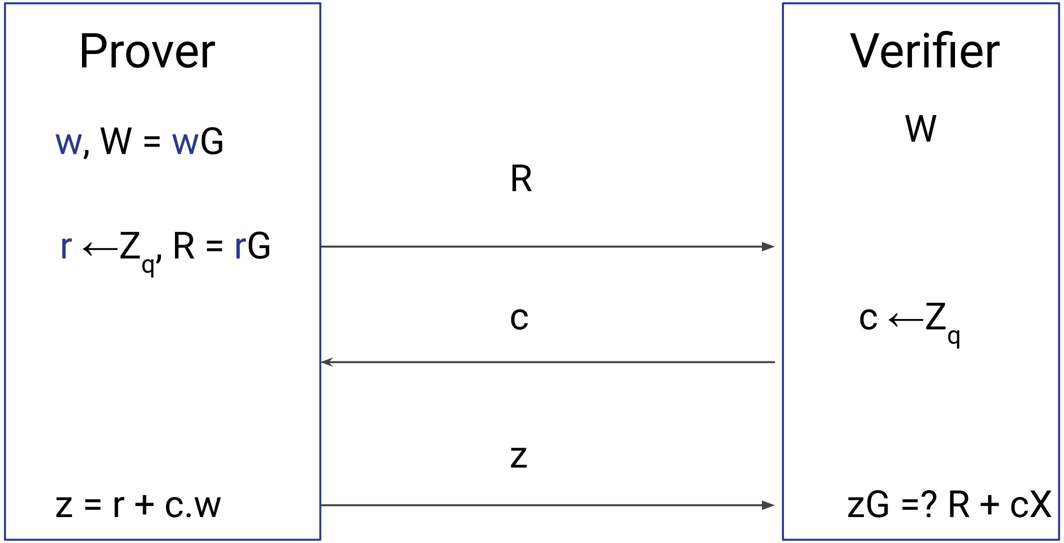
Prover wants to convince the verifier that it "*knows*" the private key w *without revealing any information* about w

Prover		Verifier
w, $W = wG$		W
$r \leftarrow Z_q$, $R = rG$	$\xrightarrow{\quad R \quad}$	
	$\xleftarrow{\quad c \quad}$	$c \leftarrow Z_q$
$z = r + c.w$	$\xrightarrow{\quad z \quad}$	$zG =? R + cW$

r is random so $r + c.w$ masks out any information related to $c.w$ or w

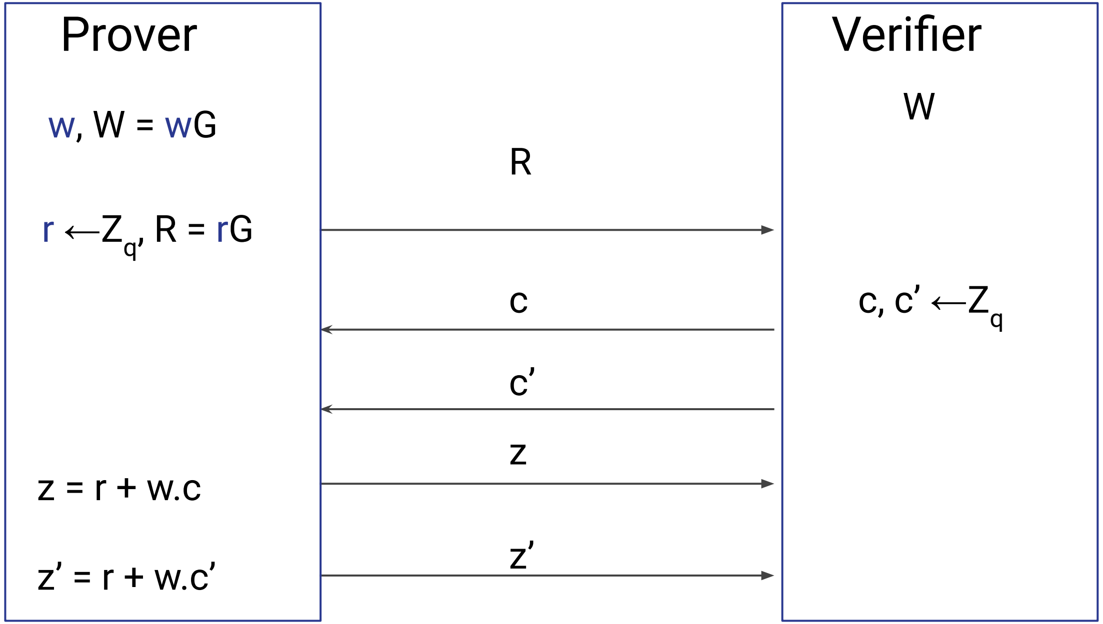
How is verifier convinced that prover "knows" w?

Prover wants to convince the verifier that it "*knows*" the private key w *without revealing any information* about w



If the verifier "*manipulates*" the prover's execution and can "extract w" from the manipulation process then the prover must have use w in its execution, i.e.„ the prover knows w

How is verifier convinced that prover "knows" w?



If the verifier "*manipulates*" the prover's execution and can "*extract w*" from the manipulation process then the prover must have use w in its execution, i.e., the prover knows w.
In this case, the verifier forces the prover to keep r *unchanged* when responding to 2 different challenges $c \neq c'$. The verifier extracts w as follow

$$z = r + wc, \; z' = r + w.c'$$
$$(z - z') = w(c - c')$$
$$w = (z - z')/(c - c')$$

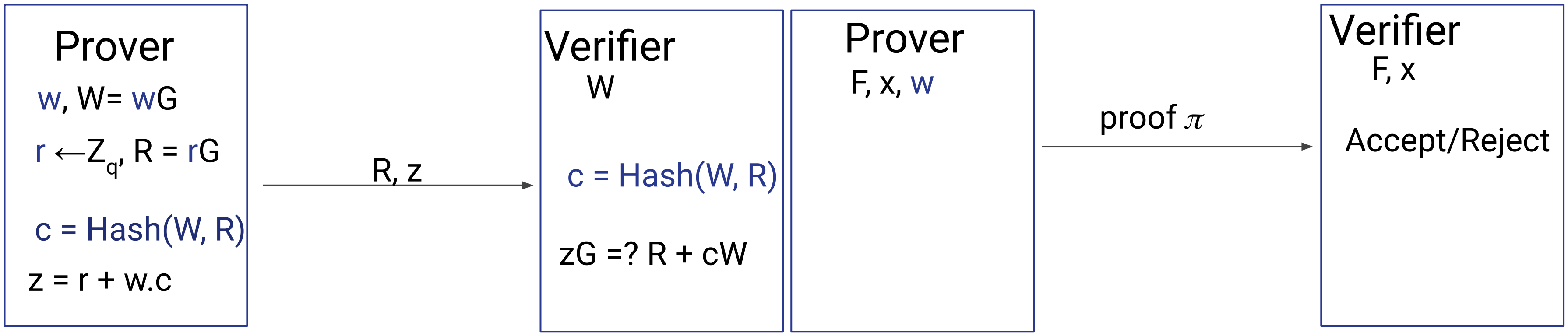
Fiat-Shamir transform

Prover		Verifier
w, $W = wG$		W
$r \leftarrow Z_q$, $R = rG$	$\xrightarrow{R}$	
	$\xleftarrow{c}$	$c \leftarrow Z_q$
$z = r + w.c$	$\xrightarrow{z}$	$zG =? R + cW$

Prover		Verifier
w, $W = wG$		W
$r \leftarrow Z_q$, $R = rG$		
$c = Hash(W, R)$		
$z = r + w.c$	$\xrightarrow{R,\ z}$	$c = Hash(W, R)$
		$zG =? R + cW$

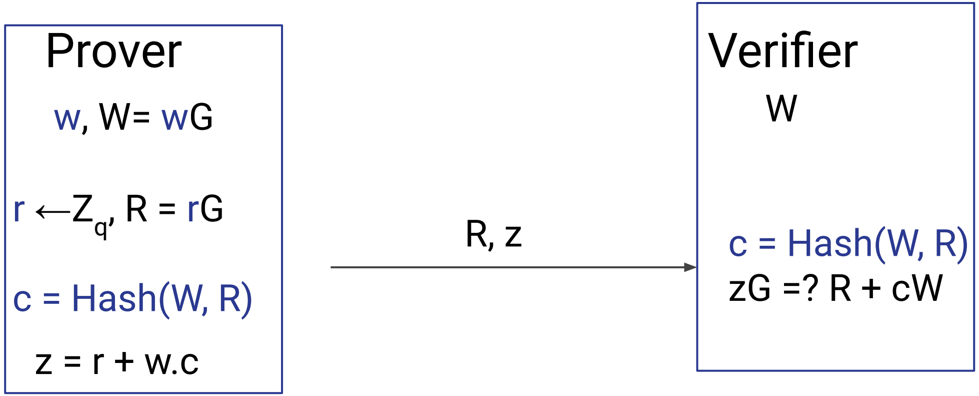
Fiat-Shamir transform: to transform *public coin* interactive protocol to an *non-interactive* one.
The challenge c is hash of transcript, i.e., *public* ZKP statement and all public values computed in the proof (e.g. commitments).
Note: the attacker can not control the output of $c = Hash(W, R)$

ZKP terminologies



- In cryptography, instead of saying proof, we use *argument* that is based on *computational* assumption (e.g. discrete log problem).
- The prover wants to convince the verifier that F(x, w) = 0 is true where x the public input, w is private input or *witness*.
- In Schorr protocol:
  - x is W, w = w
  - F(x, w) = wG - W = 0
  - proof π = (R, z)

ZKP terminologies



- If the prover convince the verifier that it “knows” w then we say that it’s “argument of knowledge”.
- If the verifier only learns that F(x, w) = 0 is correct without learning any information about the witness w then we call the protocol “zero-knowledge”.
- NARK: **N**on-interactive **AR**gument of **K**nowledge
- SNARK: **S**uccinct NARK if the proof (e.g. R, z) is short
- zk-SNARK: **zero-k**nowledge SNARK

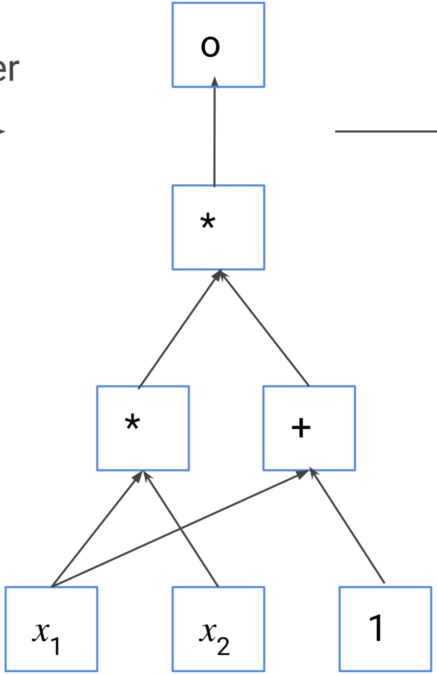
SNARK software system

Program

$o = x_1 * x_2 * (x_1 + 1)$

Compiler

Arithmetic circuit



Polynomial equations

$p_i(x) = p_0 + p_1 x + \ldots + p_d x^d$

where $p_d \in F_p$

$p_i(x) = 0$, $i= 1..n$

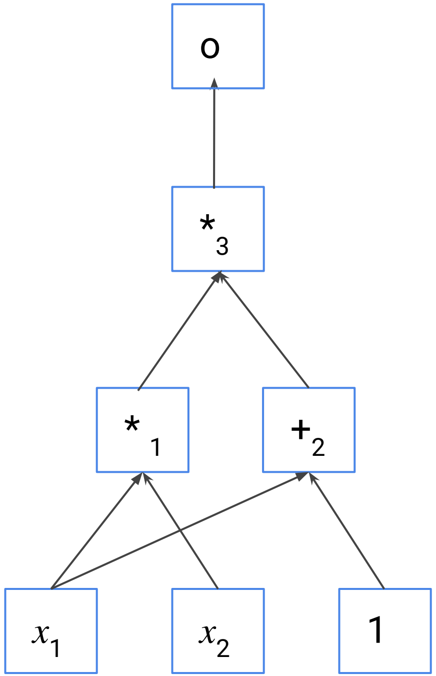
SNARK prover

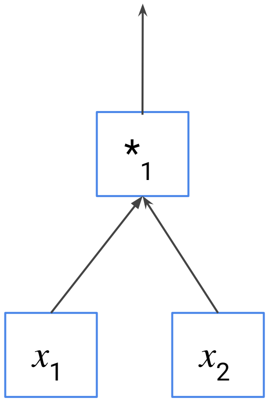
Proof π

SNARK verifier verifies π

Arithmetic circuit

Arithmetic circuit





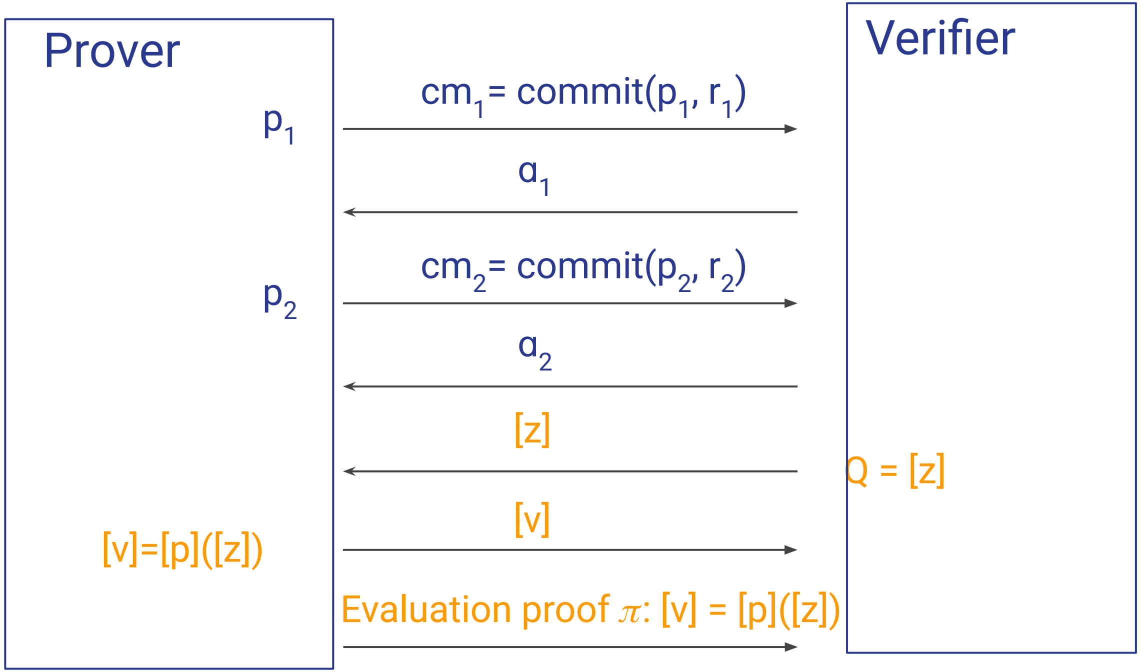
* is called a gate
Left input: x_1
Right input: x_2
Output: o_1
Constraint: $0 = \text{output}_1 - \text{left}_1 * \text{right}_1 = o_1 - x_1 * x_2$

Note that output of gate 1 (*) equals to left input of gate 3 (+)
Permutation/Copy constraint: $o_1 = \text{left}_3$

Polynomial commitment

- ❏ $p(x) = p_0 + p_1x + \dots + p_dx^d$ where $p_d \in F_p$, $p \sim 2^{256}$
- ❏ Prover $\rightarrow$ Verifier: c = commit(p, r)
- ❏ Verifier $\rightarrow$ Prover: random z
- ❏ Prover $\rightarrow$ Verifier: Evaluation proof π that $p(z) = y$

Polynomial Interactive Oracle Proof (PIOP)



Polynomial Identity Lemma

- $Pr_{z \in S}[p(z) = 0] \leqslant d/|S|$ where $p(x) = p_0 + p_1x + \ldots + p_dx^d$, $p_d \in F_p$
- Why? Polynomial $p(x)$ has at most d roots in finite field F_p.

Checking zero polynomial

- Check whether $p(x) = p_0 + p_1x + \ldots + p_dx^d \mathrel{?=} 0$ i.e. check whether all $p_i = 0$
- Verifier $\rightarrow$ Prover: random $z \in F_p$ (z is a specific value, not variable)
- Prover $\rightarrow$ Verifier: proof π that $p(z) = 0$ and if the verifier accepts the proof then $p(x) = 0$ with high probability. Why? If $p(x) \neq 0$ then the probability that $p(z) = 0$ is d/p ($d \sim$ millions, $p \sim 2^{256}$) which is negligible.

Checking multiple zero polynomials

- ❏ Check whether $f_1(x_1, \ldots, x_m) ?= 0$, $f_2(x_1, \ldots, x_m) ?= 0$, …, $f_n(x_1, \ldots, x_m) ?= 0$

Checking multiple zero polynomials (Random linear combination)

- Check whether $f_0(x_1, \ldots, x_m)$?= 0, $f_1(x_1, \ldots, x_m)$?= 0,…,$f_{n-1}(x_1, \ldots, x_m)$?= 0
- Verifier → Prover: random $r_0, \ldots, r_{n-1} \in F_p$, $p \sim 2^{256}$
- Now need to check **1** polynomial

  $f(x) = r_{0*}f_0(x_1, \ldots, x_m) + r_1{*}f_1(x_1, \ldots, x_m) + r_{n-1}{*}f_{n-1}(x_1, \ldots, x_m)$?= 0

Checking multiple zero polynomials (Random combination)

- Check whether $f_1(x_1, \ldots, x_m)$?= 0, $f_2(x_1, \ldots, x_m)$?= 0, …, $f_{n-1}(x_1, \ldots, x_m)$?= 0
- Verifier → Prover: random $r \in F_p$
- Now need to check **1** polynomial

$$f(x) = f_0(x_1, \ldots, x_m) + r * f_1(x_1, \ldots, x_m) + r^{n-1} * f_{n-1}(x_1, \ldots, x_m) \ ?= 0$$

Quotient polynomial

- Polynomial $p(x)$ has root at z iff exists quotient polynomial $q(x)$:

$$p(x) = q(x).(x - z)$$

- To check the *multiplication relation* $q(x).(x - z) ?= p(x)$, we can use pairing

Vanishing polynomial & roots of unity

- Polynomial p(x) has roots at $z_1, \ldots, z_n$ iff $\exists\, q: p(x) = q(x)(x - z_1)\ldots(x-z_n)$
- The polynomial $Z_H(x) = (x - z_1)\ldots(x - z_n)$ is called the vanishing polynomial
- If $z_i = w^i$ where $w^n = 1$ (roots of unity) then $Z_H(x) = x^n - 1$

PLONK

- ❑ zkSNARK: **z**ero-**k**nowledge **S**uccinct **N**on-interactive **AR**gument of **K**nowledge
- ❑ Polynomial commitment
- ❑ Random (linear) combination
- ❑ Fiat-Shamir transform
- ❑ Pairing

PLONK

- Pairing maps 2 points in 2 curves to finite field where G_1, G_2 are base points in 2 curves
- Notation: $[x]_1 = xG_1$, $[x]_2 = xG_2$ where x is secret that no one knows
- Important observation: $[x]_1 = xG_1$ protects the **confidentiality** of x, but the attacker can **modify** $[x]_1$

PLONK verifier

$$e([W_z]_1 + u \cdot [W_{z\omega}]_1, [x]_2) \cdot e(-(z \cdot [W_z]_1 + uz\omega \cdot [W_{z\omega}]_1 + [F]_1 - [E]_1), [1]_2) \stackrel{?}{=} 1$$

How theory guides the attack's direction?

$$e([W_z]_1 + u \cdot [W_{z\omega}]_1, [x]_2) \cdot e(-(z \cdot [W_z]_1 + uz\omega \cdot [W_{z\omega}]_1 + [F]_1 - [E]_1), [1]_2) \stackrel{?}{=} 1$$

- ❑ Which parameters that an attacker can *manipulate*?
- ❑ What is the *least effort* way to manipulate parameters?

How theory guides the attack's direction?

$$e([W_z]_1 + u \cdot [W_{z\omega}]_1, [x]_2) \cdot e(-(z \cdot [W_z]_1 + uz\omega \cdot [W_{z\omega}]_1 + [F]_1 - [E]_1), [1]_2) \stackrel{?}{=} 1$$

- ❏ $[Wz]_1$, $[Wz\omega]_1$ are under the attacker's control. To be clear, the attacker can **modify** $[Wz]_1$, $[Wz\omega]_1$ but the attacker does **not know** the inside true values Wz, Wzω
- ❏ u = hash(transcript): Fiat-Shamir transform so it's **outside the attacker's control**
- ❏ x is secret that no one knows
- ❏ F and E are computed by the verifier (not attacker) in a **complicated** multi-steps process, so let's **ignore** them (lazy attacker's mindset 😁)

How theory guides the attack's direction?

$$e([W_z]_1 + u \cdot [W_{z\omega}]_1, [x]_2) \cdot e(-(z \cdot [W_z]_1 + uz\omega \cdot [W_{z\omega}]_1 + [F]_1 - [E]_1), [1]_2) \stackrel{?}{=} 1$$

$[Wz]_1$, $[Wz\omega]_1$ are natural attack targets

How theory guides the attack's direction?

$$e([W_z]_1 + u \cdot [W_{z\omega}]_1, [x]_2) \cdot e(-(z \cdot [W_z]_1 + uz\omega \cdot [W_{z\omega}]_1 + [F]_1 - [E]_1), [1]_2) \stackrel{?}{=} 1$$

Denote

$$P[1] = [W_z]_1 + u \cdot [W_{z\omega}]_1$$
$$P[0] = -(z \cdot [W_z]_1 + uz\omega \cdot [W_{z\omega}]_1 + [F]_1 - [E]_1)$$

$$e(P[1], [x]_2) \cdot e(P[0], [1]_2) \stackrel{?}{=} 1$$

$[Wz]_1$, $[Wz\omega]_1$ are natural attack targets

How theory guides the attack's direction?

$$P[1] = [W_z]_1 + u \cdot [W_{z\omega}]_1$$

$$P[0] = -(z \cdot [W_z]_1 + uz\omega \cdot [W_{z\omega}]_1 + [F]_1 - [E]_1)$$

$$e(P[1], [x]_2) \cdot e(P[0], [1]_2) \stackrel{?}{=} 1$$

What if $[Wz]_1 = 0$, $[Wz\omega]_1 = 0$?

- ❑ $P[1] = 0 + u.0 = 0$: **neutralize** the role of Fiat-Shamir transform
- ❑ $e(P[1], [x]_2) = e(0, [x]_2) = 1$: yay!
- ❑ $P[0] = -(z.[Wz]_1 + uz\omega.[Wz\omega]_1 + [F]_1 - [E]_1) = -(z.0 + uz\omega.0 + [F]_1 - [E]_1) = -(0 + 0 + [F]_1 - [E]_1) \neq 0 \rightarrow e(P[0], [1]_2) \neq 1$
- ❑ So, the attack *doesn't* work? *No, in theory*

The attack in practice

- Sends $[Wz]_1 = 0$, $[Wz\omega]_1 = 0$ to the verifier program
- The verifier computes $e(P[0], [1]_2) = 1$ and it accepts the proof!

Security's consequence

- ❑ The prover can ***forge*** proof for **incorrect** circuit
- ❑ Even if the prover does **not** know the witness, it can convince the verifier that it knows the witness.

Why does the attack work in practice?

The attack falls through a chain of perfectly aligned software cracks

Elliptic Curve Point Representation

- Byte array: on the wire or storage
- Affine coordinate: $P = (x, y)$
- Projective coordinate: $P = (x, y, z)$. If $z \neq 0$, $P \sim (x/z, y/z, 1)$

Attack

- $[Wz]_1 = \mathcal{O} = (0, 0)$, $[Wz\omega]_1 = \mathcal{O} = (0, 0)$ where $\mathcal{O} = (0, 0)$ means its affine coordinate $(x, y) = (0, 0)$
- $P[0] \neq 0$, $P[1] = 0$

Code vulnerabilities (1)

- The verifier checks whether $[Wz]_1$, $[Wz\omega]_1$ are *on the elliptic curve* or not. $[Wz]_1 = 0$, $[Wz\omega]_1 = 0$ are *not* valid points on the curve
- The verifier does *not stop* immediately when it sees invalid points. It continues the execution, but it excludes the invalid 0 points in *some* further computations
- $[Wz]_1$, $[Wz\omega]_1$ are *included* in the crucial computation with pairing which allows the attack to work

Code vulnerabilities (2)

- Elliptic curve code *rejects* the *infinity point*
- However, according to the code, P[1] = 0 is *not infinity*. The is_point_at_infinity method checks whether the most significant bit of the P[1] is 1, but P[1] = 0's most significant bit is 0

Code vulnerabilities (3)

- Computes the inverse of 0 mod p and it *doesn't* check for **0** input.
- Uses Fermat's little theorem: x^(p — 1) = 1 mod p or x^(p — 2). x = 1 mod p or x^(p — 2) is the inverse of x mod p. When x = 0, x^(p — 2) = 0 which means that *the inverse of 0 is 0* :)

Code vulnerabilities (4)

- The array (P[0], P[1]) = (P[0] ≠ 0, P[1] = 0) are in projective coordinates (x, y, z)
- Normalization process to eliminate z (i.e. to make z = 1) → affine coordinate (x/z, y/z) ~ (x, y, 1)
- Montgomery batch inversion technique: to compute $1/x_1, \ldots, 1/x_n$, compute only **1 inversion** $I = 1/(x_1 \ldots x_n)$ and the rest are multiplication with I. For instance: $1/x_1 = x_2 \ldots x_n / (x_1 \ldots x_n) = x_2 \ldots x_n * I$.
- What will happen if I = 0? **All** $1/x_1, \ldots, 1/x_n$ are **0** although x_i may not be **0**.

Code vulnerabilities (4)

- *Batch-normalizes with Montgomery batch inversion algorithm* where P[1].z = 0 will affect P[0]. The vulnerable code outputs (P[0], P[1]) = (0, 0), i.e., it turns *non-zero point P[0]* into a 0 point.

Code vulnerabilities (4)

Before batch_normalize

P[0]:{0x12270675066dbf202e8766f5fa48648f95032fbff46996a08e05
e427ed0fffb9,0x2cce89ca786bd0a3db55776a24aa3253bce3b8ef689
849f93596b5b26afec90f,0x04ae1f4cd5f84a484acc4ba115fbd02a879
d2e30b8cd97e18f3865887213823b }

P[1]:{0x0000000000000000000000000000000000000000000000000
000000000000000,0x00000000000000000000000000000000000000
00000000000000000000000000,0x000000000000000000000000000
0000000000000000000000000000000000000 }

After batch_normalize

P[0]:{0x0000000000000000000000000000000000000000000000000
000000000000000,0x00000000000000000000000000000000000000
00000000000000000000000000,0x000000000000000000000000000
0000000000000000000000000000000000001 }

P[1]:{0x0000000000000000000000000000000000000000000000000
000000000000000,0x00000000000000000000000000000000000000
00000000000000000000000000,0x000000000000000000000000000
0000000000000000000000000000000000001 }

Code vulnerabilities (5)

- P[1] = 0 is not on the curve and P[1] = 0 is *not* infinity according to step 2, the *pairing* code considers P[1] = 0 as infinity, in the sense that e(0, R) = 1 for all R.

Thanks for your attention!