



Lezione 7 – Zero-Knowledge Proof and Blockchain

Prof. Esposito Christian

Corso di Sicurezza dei Dati



::: Sommario

- Introduzione alla ZKP e protocolli relativi;
- Applicazione nelle Blockchain

... Letture

- Sun, Xiaoqiang, et al. "A survey on zero-knowledge proof in blockchain." IEEE network 35.4 (2021): 198-205.
- Zhou, Lu, et al. "Leveraging zero knowledge proofs for blockchain-based identity sharing: A survey of advancements, challenges and opportunities." Journal of Information Security and Applications 80 (2024): 103678.
- Lavin, Ryan, et al. "A Survey on the Applications of Zero-Knowledge Proofs." arXiv preprint arXiv:2408.00243 (2024).



Introduzione a ZKP

::: Concetti Introduttivi di ZKP (1/38)

- Congettura: un'affermazione che si ritiene vera e che può essere dimostrata (ma non è ancora stata dimostrata).
- Teorema: un'affermazione che è stata dimostrata vera con una dimostrazione.
- Dimostrazione (proof): un argomento valido che dimostra che un teorema è vero.

Teorema di Pitagora

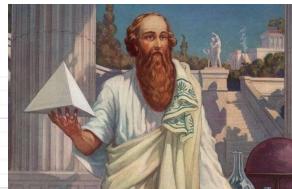
Enunciato: In ogni triangolo rettangolo il quadrato costruito sull'ipotenusa è equivalente alla somma dei quadrati costruiti sui cateti.

Dimostrazione

Hp
Triangolo Rettangolo

Th
 $Q_1 \equiv Q_2 + Q_3$

Costruiamo 2 quadrati congruenti aventi per lato la somma dei cateti

$$A_q = 4 \cdot T + Q_2 + Q_3$$
$$A_q - 4 \cdot T = Q_2 + Q_3$$
$$Q_1 \equiv Q_2 + Q_3$$


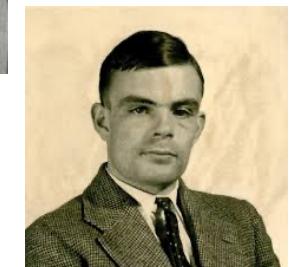
Il teorema di Gödel

- Tutte le cose vere per gli esseri umani sono anche vere (derivabili automaticamente) per le macchine? No! ([Kurt Gödel](#))?
 - Se l'uomo "sa" essere vere cose che non sono derivabili automaticamente, non è possibile avere un programma che rappresenti un cervello umano ([Roger Penrose](#)), quindi non è possibile l'Intelligenza Artificiale
 - L'argomento è piuttosto dibattuto!



TESI

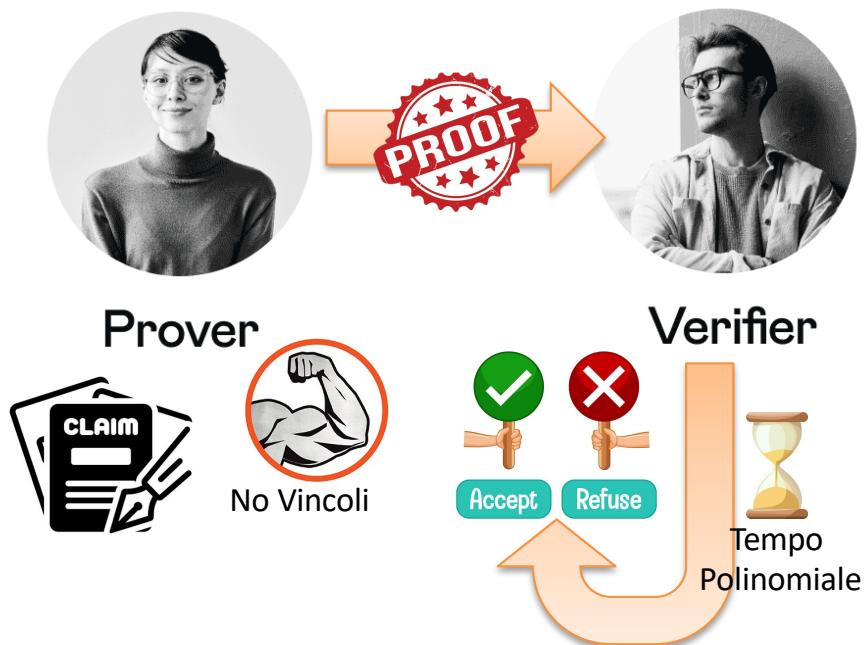
Per ogni funzione f effettivamente computabile c'è una macchina di Turing che calcola f .



Il teorema di Turing asserisce l'esistenza di problemi non decidibili, per i quali cioè non esiste alcun [algoritmo](#) in grado di dare una risposta in tempo finito su tutte le istanze del problema. La dimostrazione di questo risultato si deve ad [Alan Turing](#), che lo provò in un articolo del 1937. Il Teorema di Turing è in un certo senso la "versione informatica" del [teorema di incompletezza di Gödel](#).

::: Concetti Introduttivi di ZKP (2/38)

Consideriamo una proof come un processo interattivo tra due entità: c'è un prover e un verifier, entrambi sono degli algoritmi. Sussiste un esplicito riferimento a chi legge la proof e verifica che sia corretta.



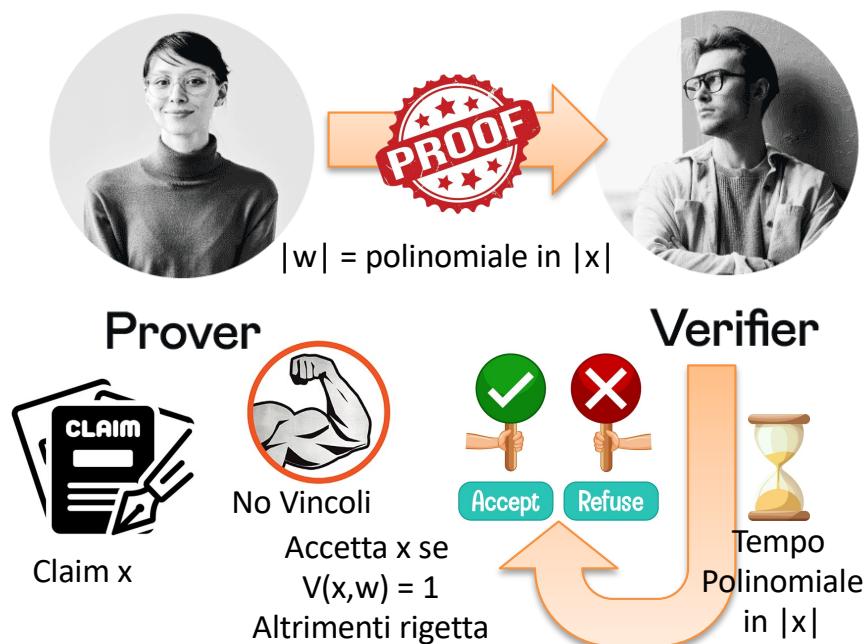
C'è un claim, che rappresenta un input sia per il prover che per il verifier. Entrambe le entità sono degli algoritmi.

Il prover invia una stringa, che rappresenta una proof, che viene letta dal verifier ed eventualmente accettata come corretta oppure rigettata. Ciò equivale ad accettare il claim come verificato o meno.

In informatica, spesso si parla in merito a proof verificabili efficientemente o NP proofs, ovvero quelle in cui la stringa che il prover invia al verifier è piccola di dimensione, e il verifier non ha molto tempo (o in maniera formale ha tempo polinomiale) per leggerla e successivamente verificarla.

::: Concetti Introduttivi di ZKP (2/38)

Consideriamo una proof come un processo interattivo tra due entità: c'è un prover e un verifier, entrambi sono degli algoritmi. Sussiste un esplicito riferimento a chi legge la proof e verifica che sia corretta.



In informatica, spesso si utilizzano proof efficientemente o NP proofs, ovvero dove in cui la stringa che il prover invia al verifier è **piccola di dimensione**, e il verifier non ha molto tempo (o in maniera formale ha tempo polinomiale) per leggerla e successivamente verificarla.

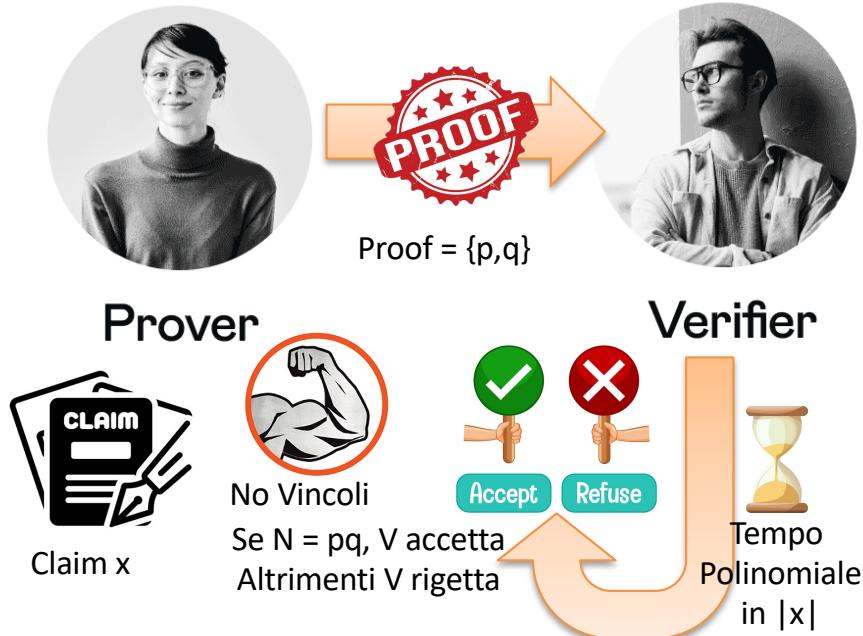
C'è un claim, che rappresenta un input sia per il prover che per il verifier. Entrambe le entità sono degli algoritmi.

Il prover invia una stringa che rappresenta il claim.

La dimensione della proof è polinomiale in base alla lunghezza del claim. Se il claim è una stringa binaria x , la proof è un messaggio codificato con una stringa binaria w . La lunghezza di w è polinomiale nella lunghezza di x , ovvero lineare, quadratica, cubica o una qualunque funzione polinomiale in $|x|$.

::: Concetti Introduttivi di ZKP (3/38)

Prendiamo in considerazione un esempio di un NP proof: il prover vuole convincere il verifier che un particolare numero N è il prodotto di due numeri primi grandi.



Questo è un claim interessante perché una procedura di verifica in tempo polinomiale di questo claim non è nota, quindi è necessario l'aiuto di un prover che costruisce un'apposita proof. Il prover non deve avere le restrizioni del verifier ed essere più «potente» del vincolo di verificare in tempo polinomiale.

La proof consiste nell'inviare i due divisori primi p e q, così che il verifier deve operare la moltiplicazione tra i due fattori, verificare se si ottiene il numero N e verificare che p e q siano dei numeri primi.

::: Concetti Introduttivi di ZKP (3/38)

Prendiamo in considerazione un esempio di un NP proof: il prover vuole convincere il verifier che un particolare numero N è il prodotto di due numeri primi grandi.



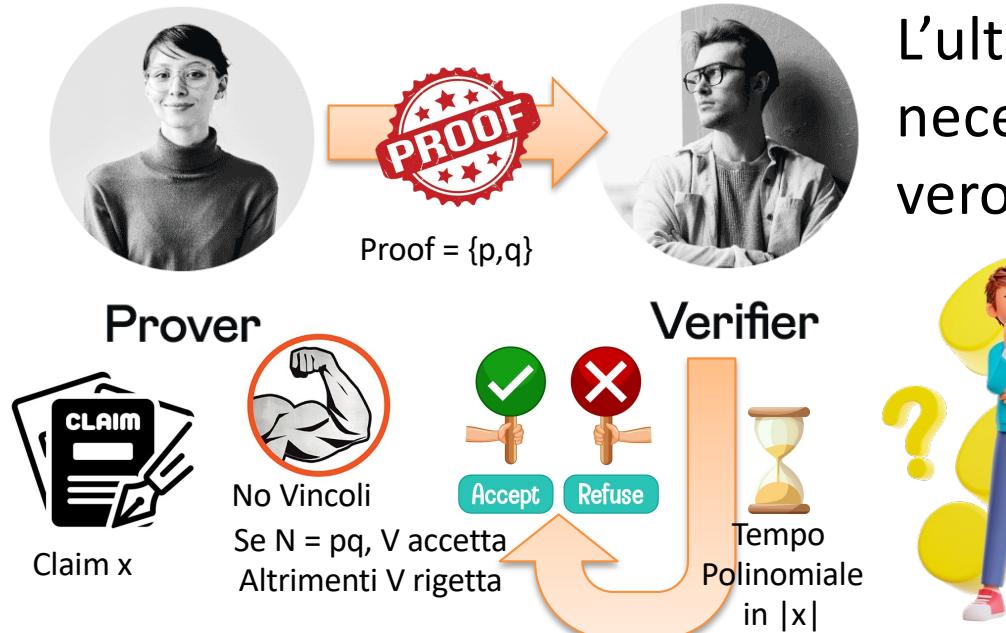
La proof consiste nell'inviare i due divisori al verifier, il verifier deve operare la **moltiplicazione** tra i due fattori, verificare se si ottiene il numero N e **verificare che p e q siano dei numeri primi**.

Questo è un claim interessante perché una procedura di verifica in tempo polinomiale di questo claim non è nota, quindi è necessario l'aiuto di un prover che costruisce un'apposita proof. Il prover non deve avere le restrizioni del verifier ed essere più «potente» del verifier.

di verificare in tempo polinomiale. Questa operazione è possibile in tempo polinomiale.

::: Concetti Introduttivi di ZKP (4/38)

A valle dell'interazione con il prover, il verifier apprende che il claim è vero, ovvero che N è il prodotto di due numeri primi, ma conosce anche qualcosa in aggiunta, ovvero che p e q sono due numeri primi.



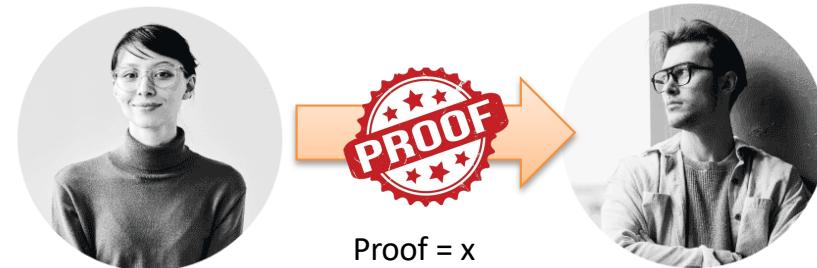
L'ultimo aspetto è qualcosa di non necessario per convincerlo che il claim è vero.



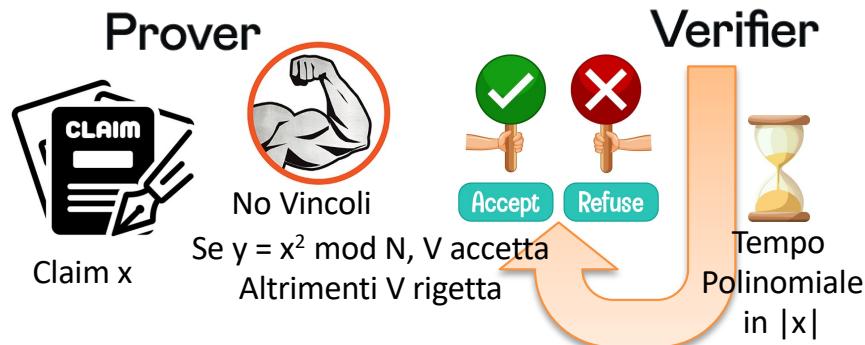
Esiste un modo alternativo di procedere? Convincere il verifier senza alcuna informazione extra?

::: Concetti Introduttivi di ZKP (4/38)

Immaginiamo un altro esempio. Il prover vuole convincere il verifier che il numero y è il residuo quadratico modulo N , ovvero che l'equazione $y = x^2 \text{ mod } N$ ammette una soluzione x che è compresa tra



1 e N , ovvero x è un numero relativamente primo a N . Non sussistono divisori in comune tra x e N .



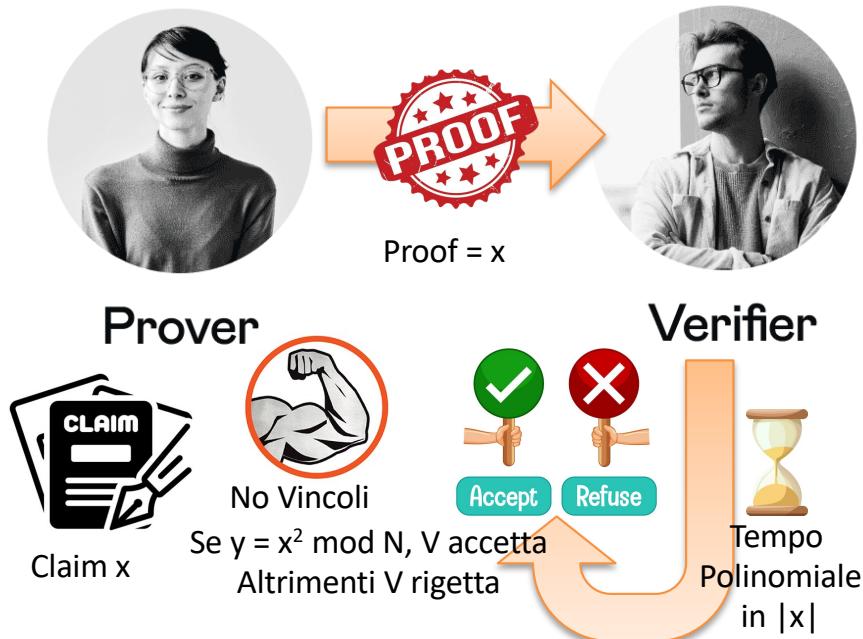
È un problema molto complesso da risolvere. Se fosse facile risolverlo in tempo polinomiale il verifier non avrebbe avuto bisogno del prover.

Questo problema è complesso come la fattorizzazione di N , che rappresenta un ben noto problema «hard» per classici computer.

Il prover è molto potente, quindi determina x e lo invia al verifier, che con questo input esegue l'equazione, ritrova y e accetta il claim.

::: Concetti Introduttivi di ZKP (5/38)

Anche dopo questa interazione, il verifier è convinto della veridicità del claim, ma conosce anche x . Quest'ultima conoscenza è aggiuntiva e non necessaria per la prova del claim.

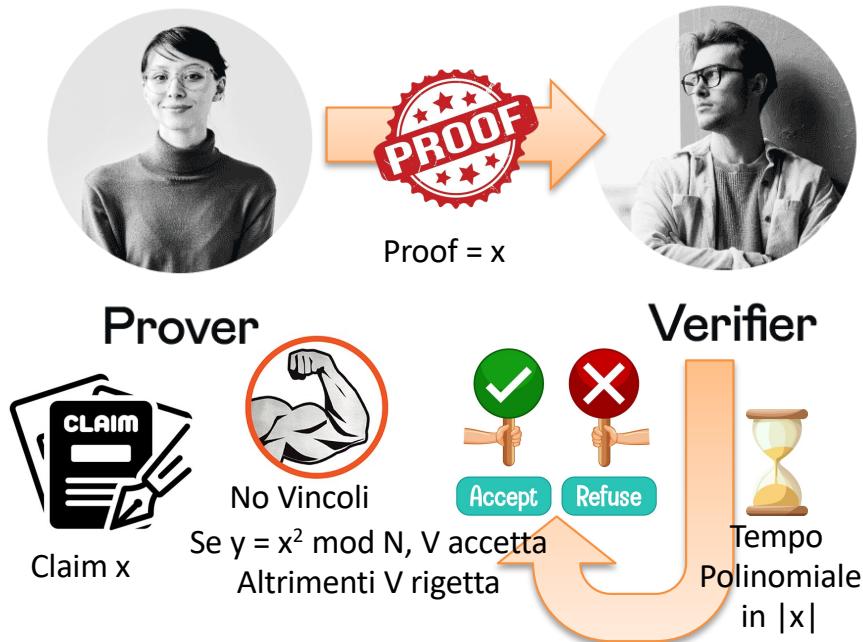


Questi esempi trattano claim molto specifici, ma in generale si è interessati a linguaggi, ovvero a dei claim che valgono in generale su un insieme di numeri: un insieme di stringhe binarie che soddisfano alcune proprietà.

Un linguaggio NP \mathcal{L} può essere definito come gli insiemi di stringhe binarie per cui esiste un verifier v che esegue in tempo polinomiale nella lunghezza del claim, espresso come stringa x , per cui sussistono due condizioni:

::: Concetti Introduttivi di ZKP (6/38)

1. Completeness [Claim veri hanno prove (corte)]: se $x \in \mathcal{L}$, esiste una proof $w \in \{0,1\}^*$ lunga secondo una funzione polinomiale in $|x|$ tale che $V(x, w) = 1$.



2. Soundness [Claim falsi non hanno proof]: se $x \notin \mathcal{L}$ esiste una proof $w \in \{0,1\}^*$, ovvero $\forall w \in \{0,1\}^*$, $V(x, w) = 0$.

Nel nostro primo esempio, \mathcal{L} è il linguaggio NP dei numeri che sono il prodotto di due numeri primi.

Se x appartiene a questo linguaggio allora esiste una proof w che convince il verifier mentre se x non appartiene al linguaggio non è possibile trovare una w che convinca diversamente il verifier, ovvero per ogni stringa che il prover invia, il verifier rigetta sempre.

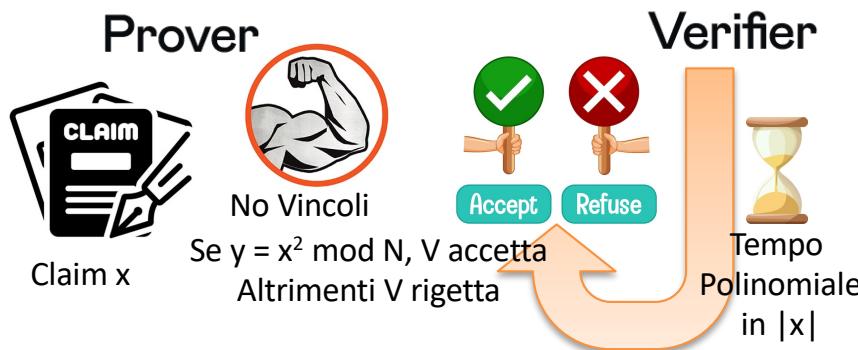
::: Concetti Introduttivi di ZKP (6/38)

1. Completeness [Claim veri hanno prove (corte)]: se $x \in \mathcal{L}$, esiste una proof $w \in \{0,1\}^*$ lunga secondo una funzione polinomiale in $|x|$ tale che $V(x, w) = 1$.



2. Soundness [Claim falsi non hanno prove accettabili]:

Prover onesti convincono sempre il verifier.

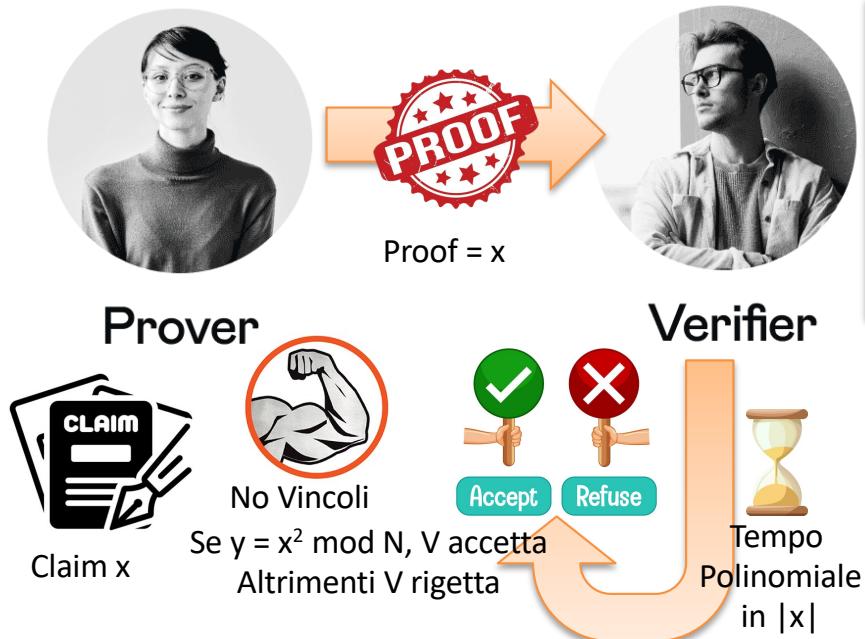


Nel nostro primo esempio, \mathcal{L} è il linguaggio NP dei numeri che sono il prodotto di due numeri primi.

Se x appartiene a questo linguaggio allora esiste una proof w che convince il verifier mentre se x non appartiene al linguaggio non è possibile trovare una w che convinca diversamente il verifier, ovvero per ogni stringa che il prover invia, il verifier rigetta sempre.

::: Concetti Introduttivi di ZKP (6/38)

1. Completeness [Claim veri hanno prove (corte)]: se $x \in \mathcal{L}$, esiste una proof $w \in \{0,1\}^*$ lunga secondo una funzione polinomiale in $|x|$ tale che $V(x, w) = 1$.



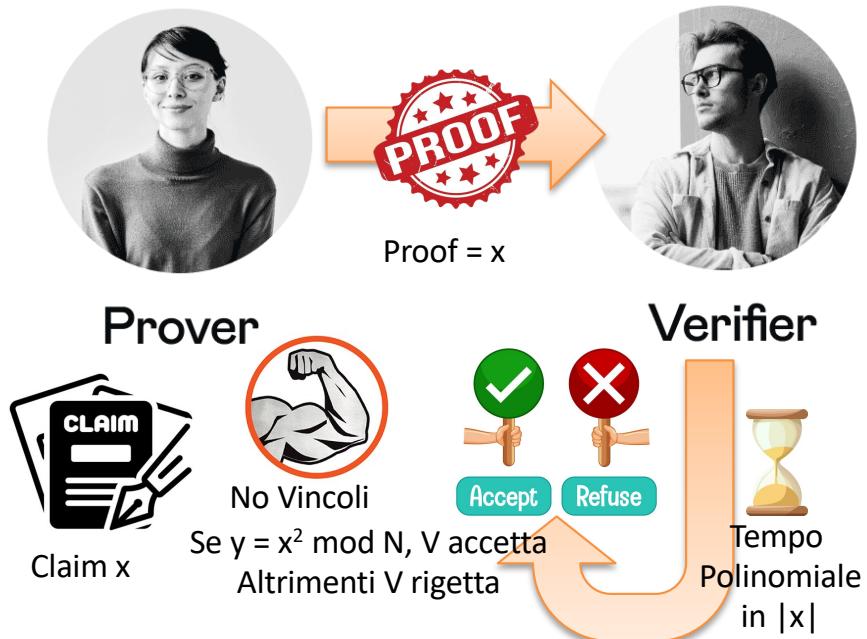
2. Soundness [Claim falsi non hanno proof]: se $x \notin \mathcal{L}$ esiste una proof $w \in \{0,1\}^*$, ovvero $\forall w \in \{0,1\}^*$, $V(x, w) = 0$.

Nei prossimi slide vedremo che indipendentemente da quello che può fare il prover non onesto, non riuscirà mai a convincere il verifier di un claim falso.

Se x appartiene a questo linguaggio allora esiste una proof w che convince il verifier mentre se x non appartiene al linguaggio non è possibile trovare una w che convinca diversamente il verifier, ovvero per ogni stringa che il prover invia, il verifier rigetta sempre.

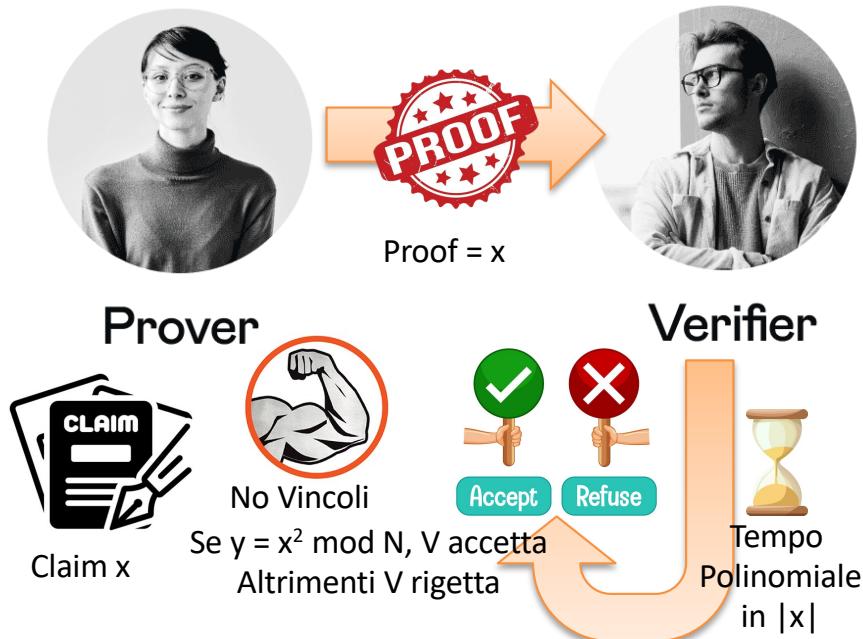
::: Concetti Introduttivi di ZKP (7/38)

Rispetto agli esempi presentati sussistono modi alternativi di convincere il verifier? Può il prover convincere un verifier che y è il residuo quadrativo modulo di N , senza mandargli la radice quadrata?



::: Concetti Introduttivi di ZKP (7/38)

Rispetto agli esempi presentati sussistono modi alternativi di convincere il verifier? Può il prover convincere un verifier che y è il residuo quadrativo modulo di N , senza mandargli la radice quadrata?



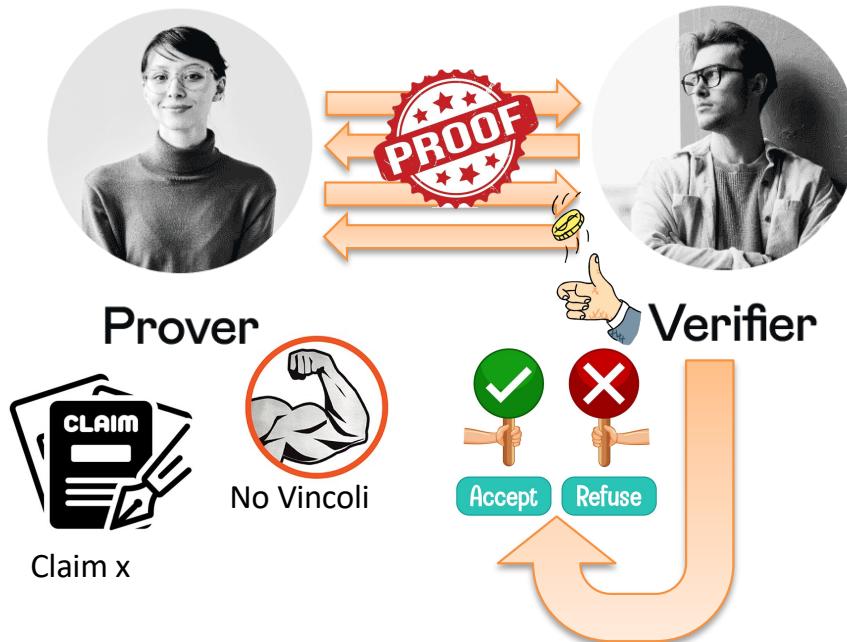
La letteratura della Zero-Knowledge Proof e di costruire delle prove che verificano il claim senza alcuna informazione aggiuntiva. In altre parole, non si presenta la soluzione ma la prova che il prover può provarla.

Il verifier non acquisisce nuova conoscenza ma solo che il claim è vero.

Questo non è possibile considerando che il prover invia una stringa w al verifier, ma il modello di prova va modificato aggiungendo due nuovi elementi: interazione e randomizzazione.

::: Concetti Introduttivi di ZKP (8/38)

Invece di avere il verifier che passivamente legge la proof e la verifica, esso è attivamente coinvolto in un'interazione non-banale con il prover. Si hanno messaggi che dal prover raggiungono il verifier e viceversa.



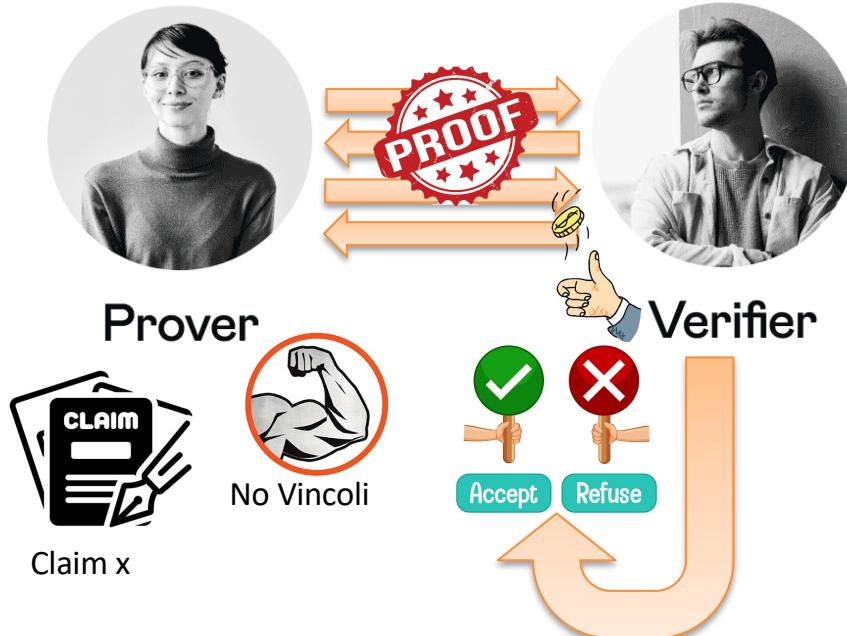
L'interazione consiste in una successione di messaggi e risposte, la cui lunghezza è un numero di coppie che è polinomiale, e maggiore di uno.

La randomizzazione implica che il verifier non è deterministico, ma lancia una moneta (come operazione primitiva) e le sue richieste al prover dipendono dall'esito di questo lancio.

Le domande del verifier sono non prevedibili, altrimenti il prover potrebbe anticiparle e inviare una sola stringa W in anticipo con tutte le risposte, ma dopo il primo messaggio del prover, il verifier ha un insieme di possibili domande e il prover deve essere in grado di rispondere a tutte.

::: Concetti Introduttivi di ZKP (9/38)

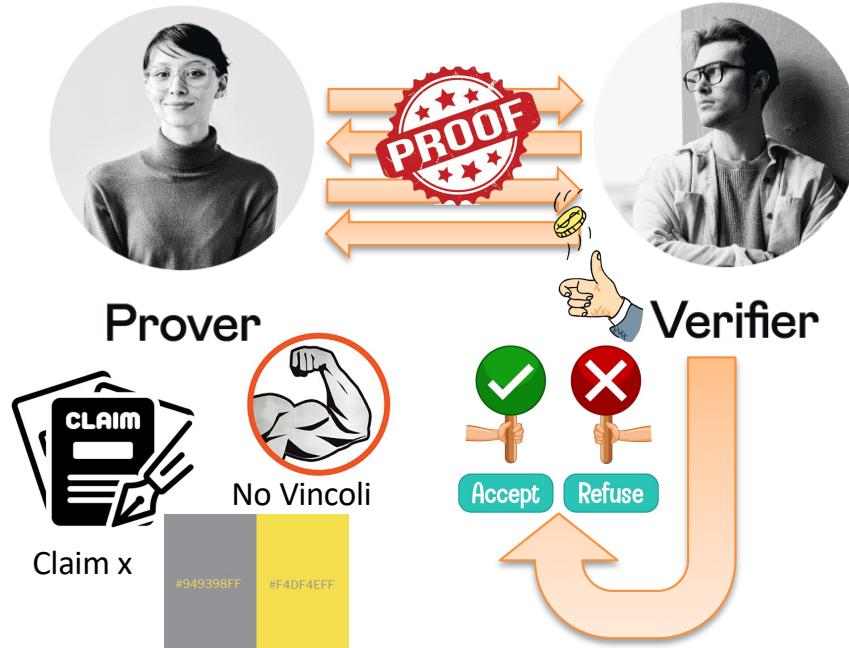
In questo modello si ammette una piccola probabilità di errore, che forse non si ha convergenza sulla verifica di un claim vero. Questo errore è piccolo e quantificabile.



Sussistono tra prover e verifier varie possibili interazioni, quindi ci sono molte possibili esecuzioni del protocollo. Il verifier conserva la memoria delle interazioni e alla fine eseguire un algoritmo e prendere una decisione.

::: Concetti Introduttivi di ZKP (9/38)

In questo modello si ammette una piccola probabilità di errore, che forse non si ha convergenza sulla verifica di un claim vero. Questo errore è piccolo e quantificabile.

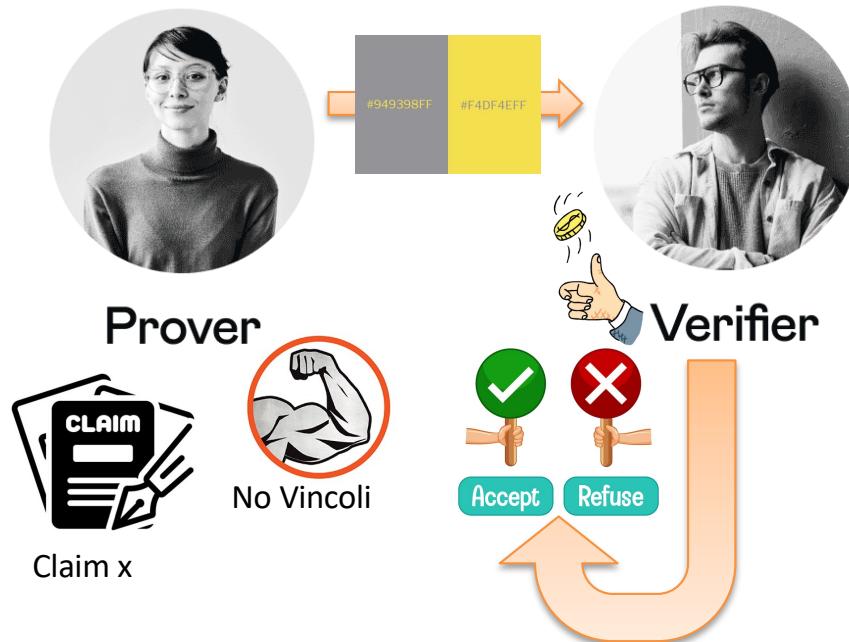


Sussistono tra prover e verifier varie possibili interazioni, quindi ci sono molte possibili esecuzioni del protocollo. Il verifier conserva la memoria delle interazioni e alla fine eseguire un algoritmo e prendere una decisione.

Vediamo un primo esempio: il claim è che una particolare pagina ha due colori invece di uno solo. Il prover è in grado di vedere i colori mentre il verifier è daltonico. Il lavoro del prover è convincere il verifier che questa specifica pagina ha due colori, senza mostrarglieli.

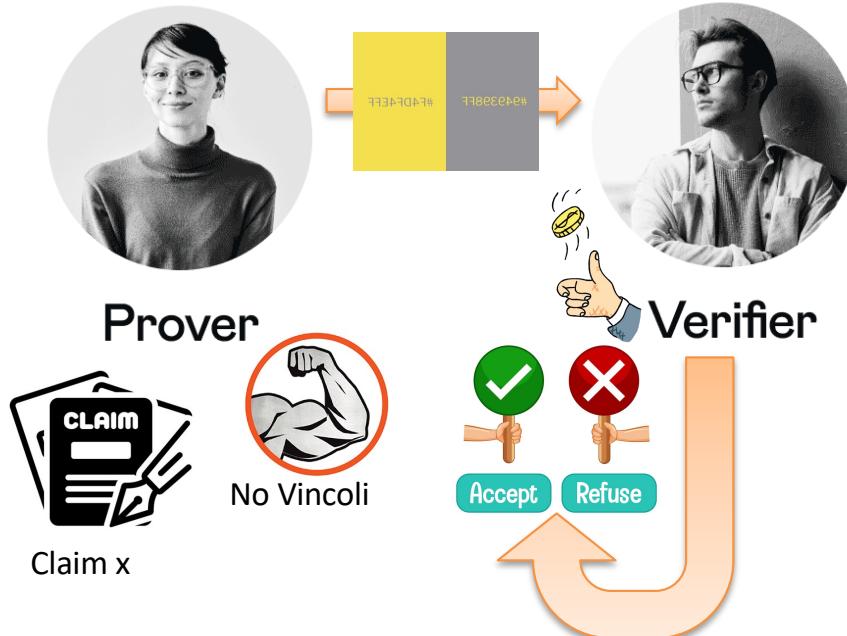
::: Concetti Introduttivi di ZKP (10/38)

Il prover manda la pagina al verificatore, che non può vederla, ma lancia una moneta: se esce testa allora chiede di ruotare la pagina, altrimenti di non fare nulla.



::: Concetti Introduttivi di ZKP (10/38)

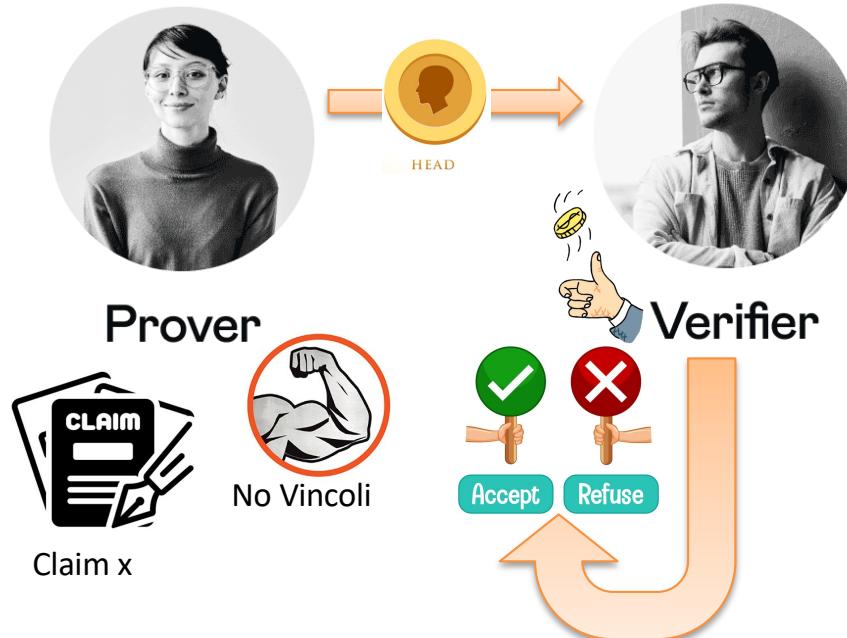
Il prover manda la pagina al verificatore, che non può vederla, ma lancia una moneta: se esce testa allora chiede di ruotare la pagina, altrimenti di non fare nulla.



Immaginiamo esca testa e il verifier gira la pagina e la rimanda al prover, che vede la pagina. Il prover ricorda la pagina prima e vede la pagina a valle della modifica fatta. Vedendo i colori può determinare se la moneta del verifier ha restituito testa o croce.

::: Concetti Introduttivi di ZKP (10/38)

Il prover manda la pagina al verificatore, che non può vederla, ma lancia una moneta: se esce testa allora chiede di ruotare la pagina, altrimenti di non fare nulla.

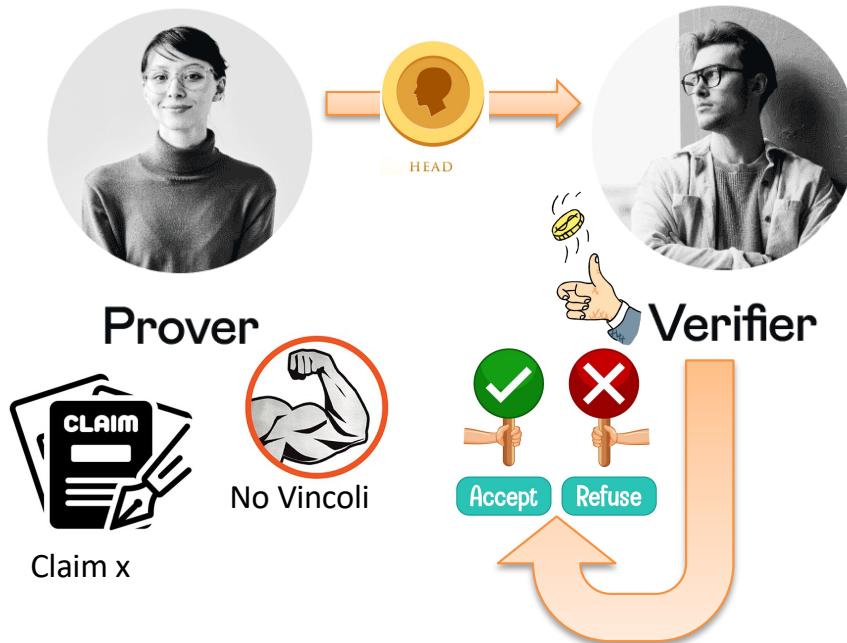


Immaginiamo esca testa e il verifier gira la pagina e la rimanda al prover, che vede la pagina. Il prover ricorda la pagina prima e vede la pagina a valle della modifica fatta. Vedendo i colori può determinare se la moneta del verifier ha restituito testa o croce.

Il prover risponde che la moneta ha restituito testa. Così, il verifier confronta il risultato restituito da quello di prima. Se corrispondono allora il prover è in grado di dire se la pagina è stata girata o meno, e questo è possibile se effettivamente ci sono due colori.

::: Concetti Introduttivi di ZKP (11/38)

Se la pagina ha due colori e il verifier accetta sempre quando la risposta corrisponde all'uscita della moneta, allora ho completeness. Il prover potrebbe imbrogliare, quando la pagina è monocromatica, provando ad indovinare, avendo il 50% delle possibilità di indovinare correttamente.



Nel caso di pagine monocromatiche, la probabilità che il verifier accetti il claim è meno dei 0,5.

È possibile ripetere questo processo varie volte (di numero k), così da avere una probabilità di accettazione per claim falsi non superiore a $1/2^k$.

Quindi, la probabilità di accettare claim falso è molto piccola al crescere di k ma non nulla.

::: Concetti Introduttivi di ZKP (12/38)

Facciamo l'esempio di natura matematica di prima. Immaginiamo di avere la coppia N e y tale che esista un numero x tale che $y = x^2 \bmod N$. Il prover prova a convincere il verifier che la coppia N,y è un membro del linguaggio \mathcal{L} delle coppie di numeri che soddisfano il claim.



Prover

Ciò avviene senza che il prover debba inviare la prova banale contenente x .

::: Concetti Introduttivi di ZKP (12/38)

Facciamo l'esempio di natura matematica di prima. Immaginiamo di avere la coppia N e y tale che esista un numero x tale che $y = x^2 \bmod N$. Il prover prova a convincere il verifier che la coppia N,y è un membro del linguaggio \mathcal{L} delle coppie di numeri che soddisfano il claim.



Prover

Sceglie un valore casuale
 $1 \leq r \leq N$ tale che
 $\text{MCD}(r,N) = 1$

$$s = r^2 \bmod N$$



Verifier

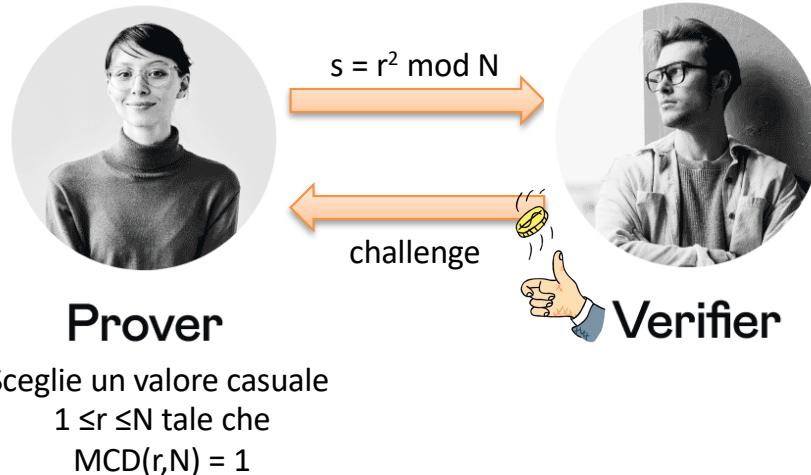
Ciò avviene senza che il prover debba inviare la prova banale contenente x .

Il prover sceglie un valore casuale r tra 1 e N , tale che non abbiamo nessun divisore in comune con N .

Il prover invia il numero s ottenuto dal quadrato di r modulo N . Inoltre, sussiste che se gli si invia la radice quadrata di s e quella di s moltiplicato per y , con y l'input, allora il verifier è convinto che il claim è vero, perché se s ha una radice quadrata e y volte s ha una radice quadrata, allora anche y ha una radice quadrata modulo N .

::: Concetti Introduttivi di ZKP (12/38)

Questo modo di procedere non è quello che cerchiamo: inviare la radice quadrata di s equivale a inviare r e la radice quadrata di s per y consiste in r per x , allora il verifier conoscerebbe la radice quadrata di y .



Non si vuole mandare i due numeri per non consentire al verifier di acquisire la conoscenza extra. Il prover manderà solo una delle due radici quadrate, e sarà il verifier a decidere quale.

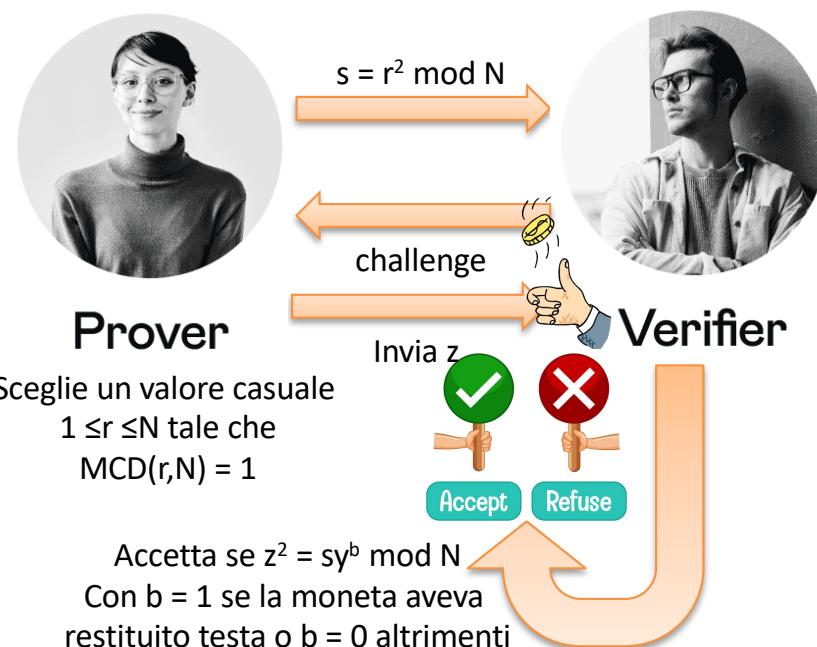
Il verifier decide quale ricevere lanciando una moneta, e decidere per un'equazione che il prover non può risolvere. Il fatto che il prover può eseguire e risolvere le due equazioni equivale al fatto che è onesto e il claim è vero.

Il verifier lancia una moneta e sceglie la challenge di ottenere r con testa o $r\sqrt{y} \bmod N$.

::: Concetti Introduttivi di ZKP (13/38)

Il verifier controlla se ha avuto la risposta giusta.

- Se il claim è vero, allora il verifier lo accetterà



- Se il claim è falso, il prover disonesto avrà non più del 50% di probabilità di ingannare il verifier. Questo perché sa risolvere solo una delle due equazioni, ma non entrambe.

Il prover deve solo conoscere $x = \sqrt{y}$.

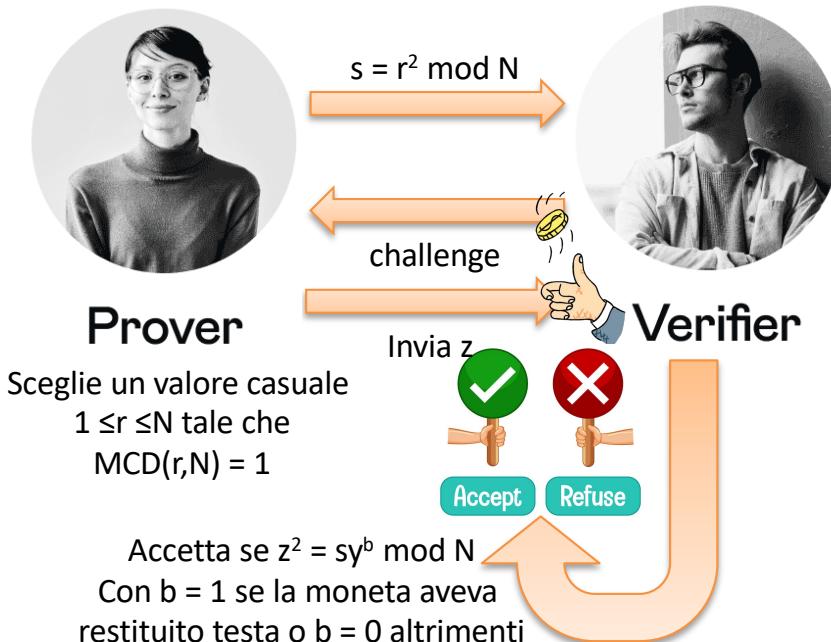
Ripetendo le interazioni, la probabilità di accettazione di claim falsi si riduce.

Il verifier non viene mai a conoscenza di x , ma di un numero random r , oppure di un multiplo random di x . Il prover invia, quindi, numeri random che sono indipendenti tra loro (all'inizio di ogni ripetizione si sceglie un numero random).

::: Concetti Introduttivi di ZKP (14/38)

(P,V) è una proof interattiva per \mathcal{L} , se V è probabilistica in tempo polinomiale ($\text{poly}(|x|)$) e sussistono queste due condizioni:

1. Completeness: se $x \in \mathcal{L}$, V accetta sempre.

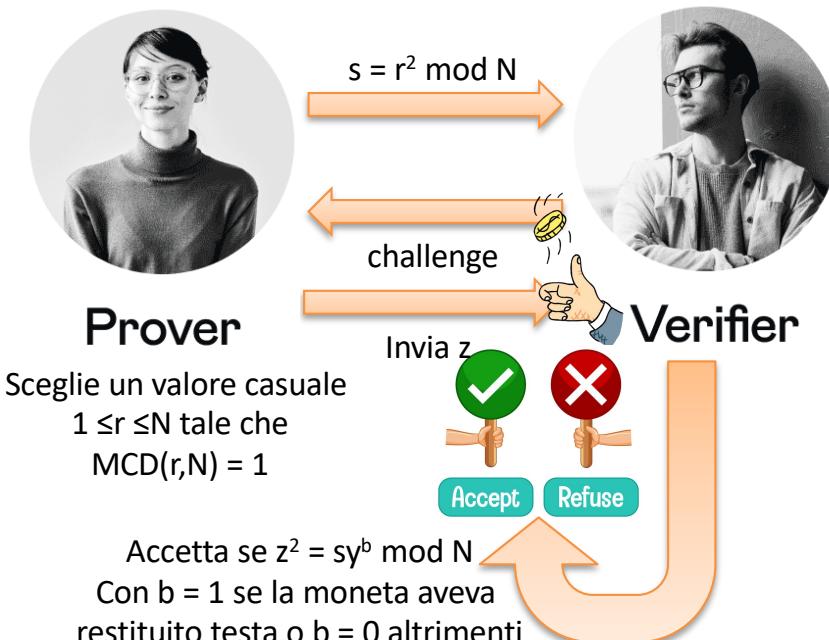


2. Soundness: se $x \notin \mathcal{L}$, per ogni strategia di un prover non onesto (che lo aiuterà a convincere il verifier), V non accetterà ad eccezione di un caso a probabilità trascurabile.

::: Concetti Introduttivi di ZKP (14/38)

(P,V) è una proof interattiva per \mathcal{L} , se V è probabilistica in tempo polinomiale ($\text{poly}(|x|)$) e sussistono queste due condizioni:

1. Completeness: se $x \in \mathcal{L}$, $\Pr[(P,V)(x) = \text{accept}] = 1$.

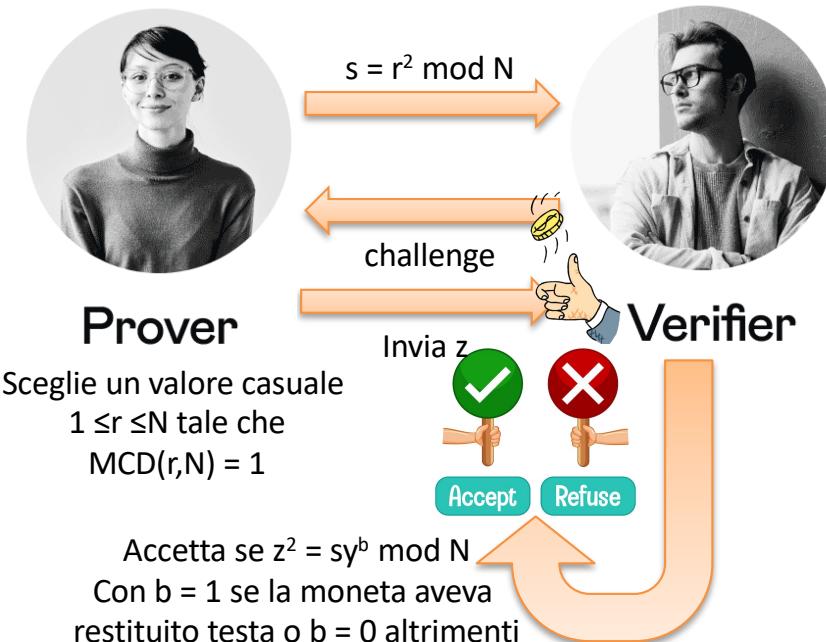


2. Soundness: se $x \notin \mathcal{L}$, per ogni P^* , $\Pr[(P^*,V)(x) = \text{accept}] = \text{negl}(|x|)$, dove $\text{negl}(\lambda) < 1/\text{polinomiale}(\lambda)$ per tutte le funzioni polinomiali (una funzione che cresce più lentamente di ogni altra funzione polinomiale).

::: Concetti Introduttivi di ZKP (14/38)

(P,V) è una proof interattiva per \mathcal{L} , se V è probabilistica in tempo polinomiale ($\text{poly}(|x|)$) e sussistono queste due condizioni:

1. Completeness: se $x \in \mathcal{L}$, $\Pr[(P,V)(x) = \text{accept}] \geq c$.



2. Soundness: se $x \notin \mathcal{L}$, per ogni P^* , $\Pr[(P^*,V)(x) = \text{accept}] \leq s$.

Con la condizione che $c - s \geq 1/\text{poly}(|x|)$, per l'equivalenza alle definizioni precedenti.

Nella realtà bastano queste tdisuguaglianze.

Un interactive preferred (IP) language è un linguaggio \mathcal{L} per cui sussiste una interactive proof, mentre un linguaggio NP è quello per cui sussiste una proof NP. L'esecuzione del prover può essere lenta, ma il verifier deve essere in tempo polinomiale probabilistico.

::: Concetti Introduttivi di ZKP (15/38)

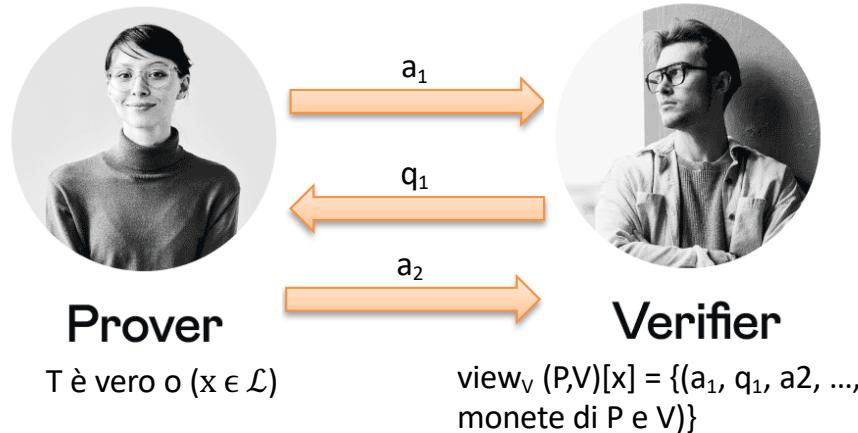
Una zero-knowledge interactive proof è quel tipo di prova che ogni volta che l'affermazione è vera, allora il verifier passa attraverso una dimostrazione interattiva di un prover, e alla fine è convinto con alta probabilità che l'affermazione sia vera. Durante l'interazione, il verifier acquisisce una cronologia di interazioni, il risultato delle monete che ha lanciato, ma quello che ha computato dopo l'interazione è fondamentalmente lo stesso di quello che avrebbe elaborato prima dell'interazione.

La dimostrazione interattiva non ha incrementato il potere computazione del verifier o gli ha consentito di ottenere più di quello che avrebbe determinato prima.

Questa proprietà non è valida solo per un verifier onesto ma per ogni classe di verifier anche quello malizioso.

::: Concetti Introduttivi di ZKP (16/38)

Il verifier è caratterizzato da una vista, ovvero la trascrizione delle risposte e domande intercorse durante l'interazione e dei risultati delle monete lanciate.

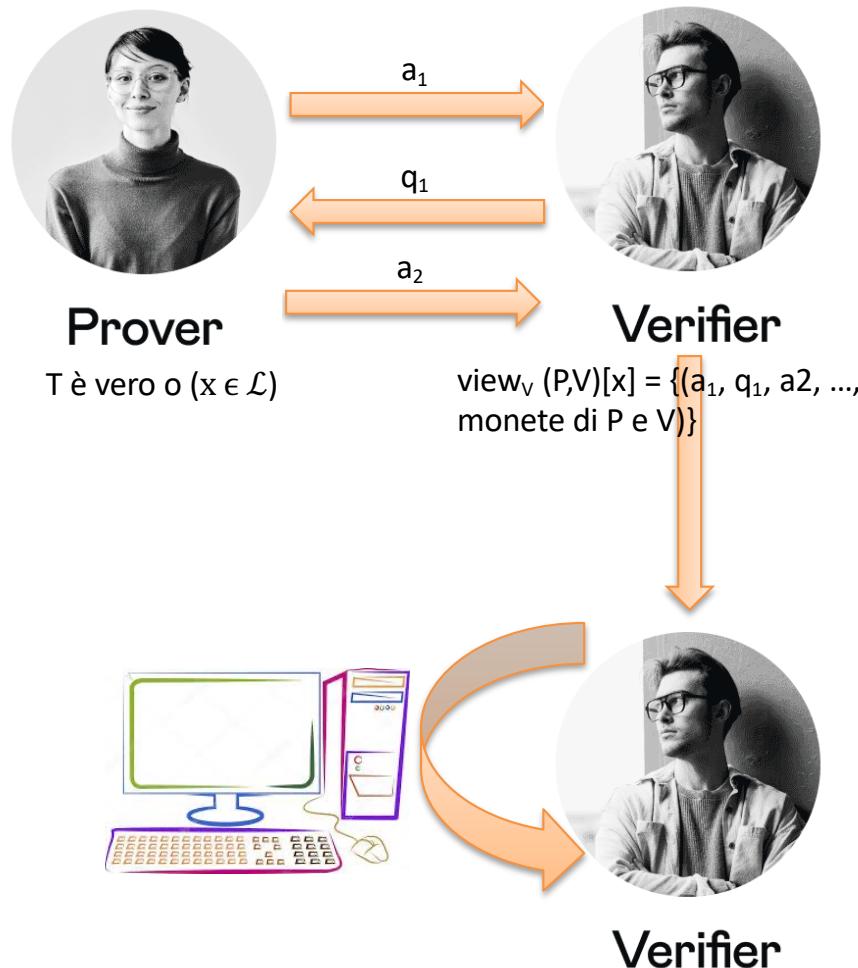


La vista di V non fornisce nulla di nuovo al verifier se e solo se esso avesse simulato l'interazione e la vista simulata e quella reale sono computazionalmente indistinguibili (simulation paradigm).

Se è disponibile un comparatore a tempo polinomiale e questo non è in grado di determinare se un campione appartiene alla vista reale o quella simulata, allora le due viste sono computazionalmente indistinguibili.

::: Concetti Introduttivi di ZKP (16/38)

Il verifier è caratterizzato da una vista, ovvero la trascrizione delle risposte e domande intercorse durante l'interazione e dei risultati delle monete lanciate.



La vista di V non fornisce nulla di nuovo al verifier se e solo se esso avesse simulato l'interazione e la vista simulata e quella reale sono computazionalmente indistinguibili (simulation paradigm).

Se è disponibile un comparatore a tempo polinomiale e questo non è in grado di determinare se un campione appartiene alla vista reale o quella simulata, allora le due viste sono computazionalmente indistinguibili.

::: Concetti Introduttivi di ZKP (17/38)

Se consideriamo le due viste come due distribuzioni di stringhe di k-bit, essi sono computazionalmente indistinguibili se, per ogni algoritmo di distinzione in tempo polinomiale D , dato un numero polinomiale di campioni da D_b , la probabilità di indovinare la distribuzione D_b tra le due di partenza è pari a $\frac{1}{2}$ più una costante trascurabile, funzione di k .

$$\text{Prob}[D \text{ indovini } b] < \frac{1}{2} + \text{negl}(k)$$

Un protocollo interattivo (P,V) è zero-knowledge per un linguaggio \mathcal{L} , se esiste un simulatore probabilistico a tempo polinomiale tale che per ogni $x \in \mathcal{L}$, le seguenti distribuzioni di probabilità sono indistinguibili a tempo polinomiale:

- $\text{view}_V(P,V)[x] = \{(a_1, q_1, a2, \dots, \text{monete di } P \text{ e } V)\}$
- $\text{Sim}(x, 1^\lambda)$

Non si è caratterizzato il verifier, quindi la definizione vale per ogni tipo.

::: Concetti Introduttivi di ZKP (18/38)

Definizione precedente: Un protocollo interattivo (P,V) è zero-knowledge con un verificatore onesto per un linguaggio \mathcal{L} , se esiste un simulatore probabilistico a tempo polinomiale Sim tale che per ogni $x \in \mathcal{L}$, le seguenti distribuzioni di probabilità sono indistinguibili a tempo polinomiale:

- $\text{view}_V(P,V)[x] \approx \text{Sim}(x, 1^\lambda)$

Definizione corretta: Un protocollo interattivo (P,V) è zero-knowledge per un linguaggio \mathcal{L} , se per ogni verifier probabilistico a tempo polinomiale V^* esiste un simulatore a tempo polinomiale simulatore tale che per ogni $x \in \mathcal{L}$, le seguenti distribuzioni di probabilità sono indistinguibili a tempo polinomiale:

- $\text{view}_{V^*}(P,V^*)[x] \approx \text{Sim}(x, 1^\lambda)$

Ovvero che la vista reale non aggiunge nessun ulteriore potere computazionale, indipendentemente da come è implementato il verifier.

::: Concetti Introduttivi di ZKP (19/38)

Nel caso in cui le due distribuzioni siano computazionalmente indistinguibili si ha computational zero-knowledge (CZK):

- $\text{view}_V(P,V)[x] \approx \text{Sim}(x, 1^\lambda)$

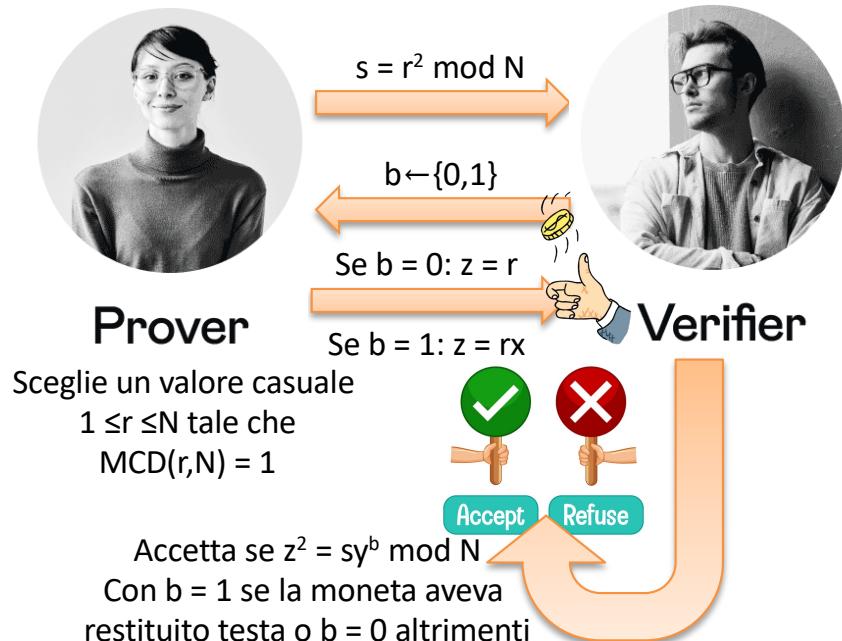
Nel caso in cui le due distribuzioni siano identiche si ha perfect zero-knowledge (PZK):

- $\text{view}_V(P,V)[x] = \text{Sim}(x, 1^\lambda)$

Nel caso in cui le due distribuzioni siano statisticamente simili (o vicine) si ha Statistical zero-knowledge (SZK).

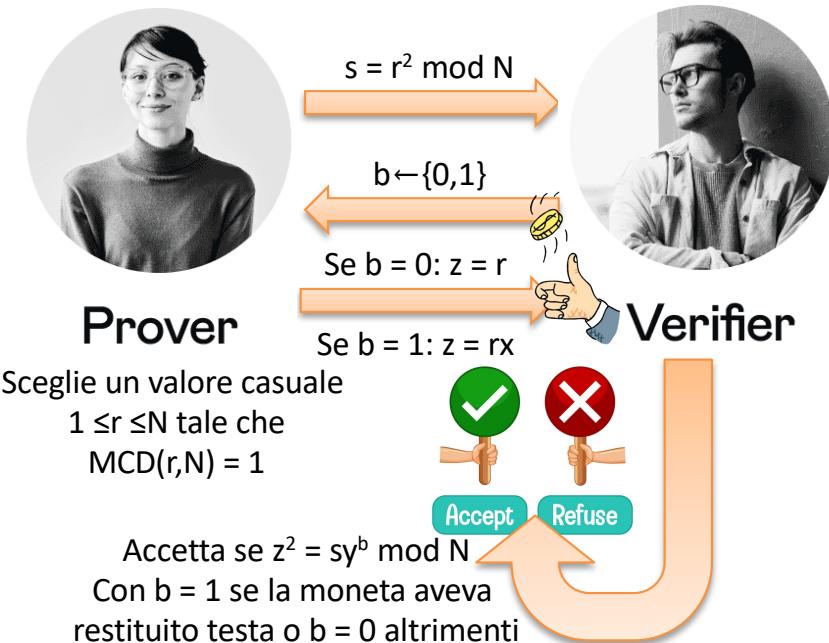
CZK si ottiene perché alcune elaborazioni sono complesse e/o richiedono tempo, per questo si ha ZK; mentre nel caso PZK questa non dipende da nessuna assunzione preliminare ma è una condizione universale.

::: Concetti Introduttivi di ZKP (20/38)



Qual è la vista di questo verifier?

::: Concetti Introduttivi di ZKP (20/38)

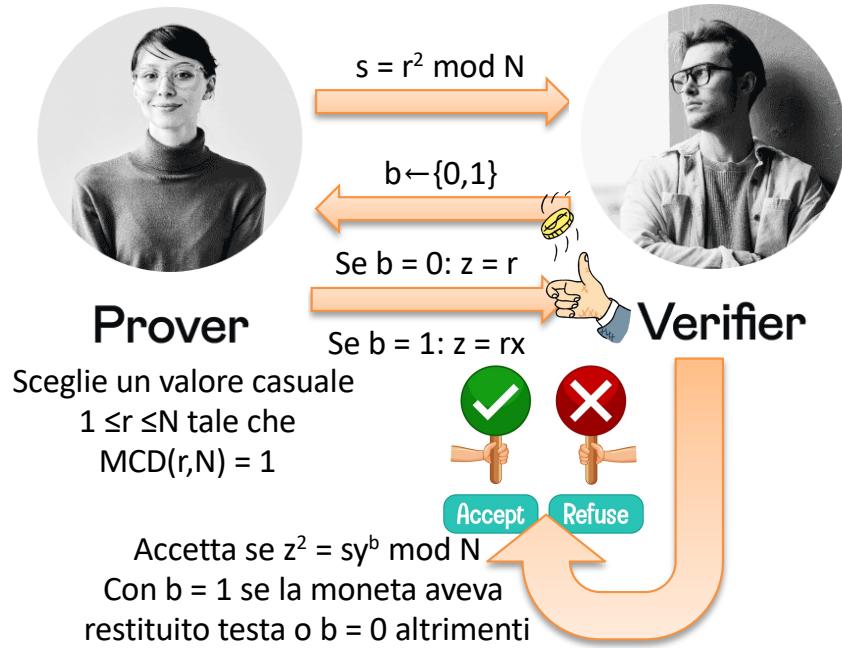


Qual è la vista di questo verifier?

$\text{view}_V(P,V)[x]:(s,b,z)$

Come è possibile simulare questo con un Sim PPT?

::: Concetti Introduttivi di ZKP (20/38)



Qual è la vista di questo verifier?

$\text{view}_V(P,V)[x]:(s,b,z)$

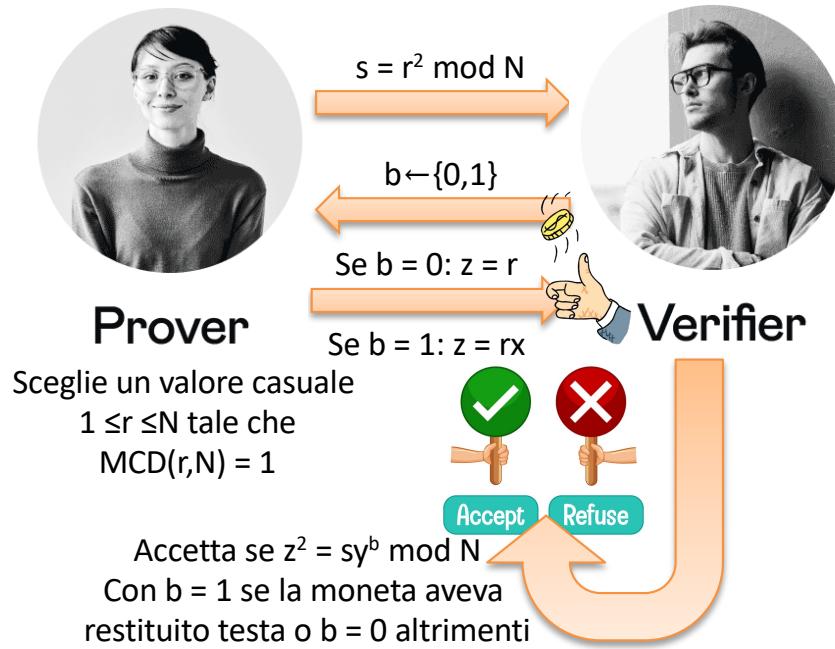
Come è possibile simulare questo con un Sim PPT?

Invece di procedere nell'ordine del protocollo presentato, ma si inizia dalla richiesta del verifier, scegliendo un bit b a caso.

Successivamente si considera un numero random $z \in \mathbb{Z}_N^*$, adesso bisogna determinare la s per cui la z è una giusta risposta. Pertanto si sceglie $s = z^2/y^b$, in base all'equazione di accettazione del verifier.

La tripla ottenuta è computazionalmente distribuita come la vista del verifier.

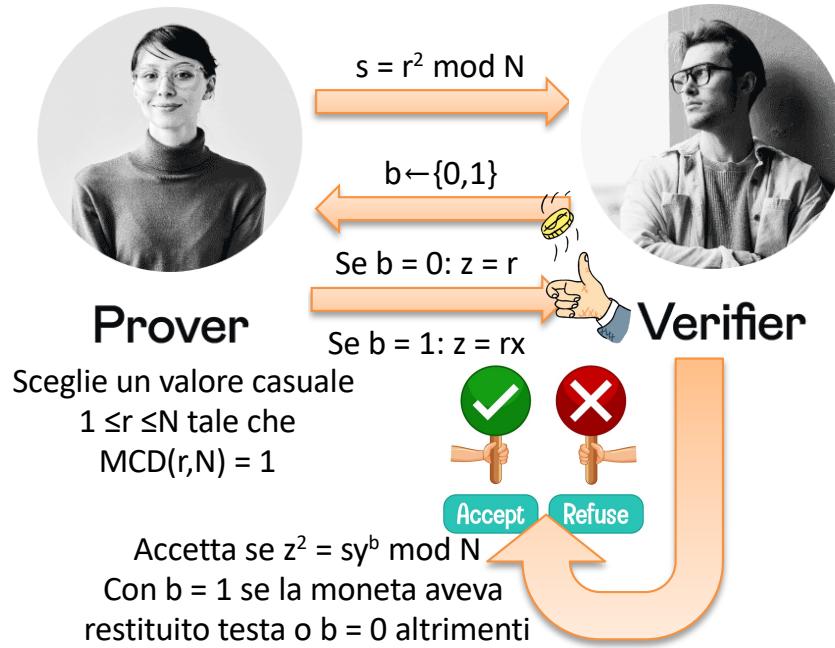
::: Concetti Introduttivi di ZKP (21/38)



Il simulatore non convince di nessun claim, ma ricostruisce un campione di una distribuzione computazionalmente identica alla vista reale del protocollo.

Cosa succede se il verifier non è onesto? Ovvero che non lancia la moneta ma possa fare una operazione molto complessa sulla s ricevuta?

::: Concetti Introduttivi di ZKP (21/38)

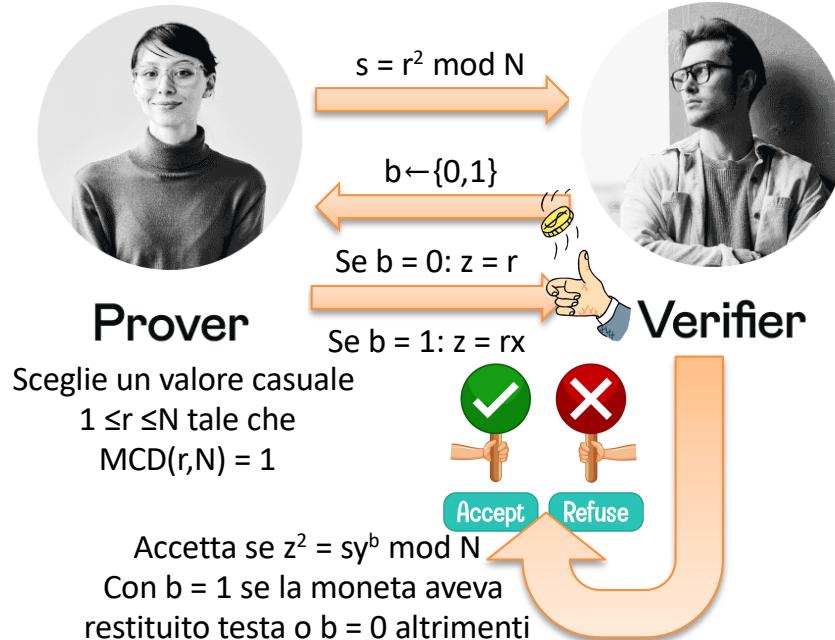


Il simulatore non convince di nessun claim, ma ricostruisce un campione di una distribuzione computazionalmente identica alla vista reale del protocollo.

Cosa succede se il verifier non è onesto? Ovvero che non lancia la moneta ma possa fare una operazione molto complessa sulla s ricevuta?

Si inizia scegliendo un bit b a caso, si considera un numero random $z \in \mathbb{Z}_N^*$, e si determina la s per cui la z è una giusta risposta. Pertanto si sceglie $s = z^2/y^b$, in base all'equazione di accettazione del verifier. A questo punto si fornisce il tutto al verifier V^* che opera le sue operazioni. Se $V^*(N, Y, S) = b$ si rilascia la tripletta, altrimenti si ricomincia. In due iterazioni è possibile ottenere (s, b, z) della vista reale.

::: Concetti Introduttivi di ZKP (22/38)



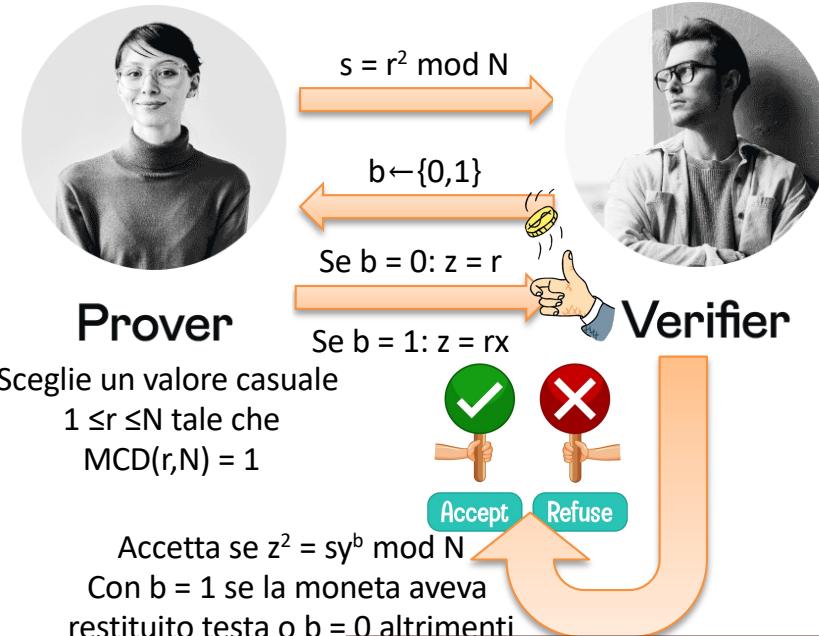
In un ZKP, si ha non solo che il prover convince il verifier della correttezza del claim, ma anche che conosce il valore x , senza comunicarlo al verifier.

Si consideri $L_R = \{x : \exists w \text{ tale che } R(xw) = \text{accept}\}$ per la relazione R a tempo polinomiale.

(P,V) è una proof of knowledge (POK) per L_R se esiste un algoritmo di estrazione (della conoscenza) E probabilistico a tempo polinomiale tale che $\forall x \in L_R$ in tempo polinomiale $E^P(x)$ ritorna w tale che $V(x, w) = \text{accept}$, con $E^P(x)$ che indica che E esegue P ripetutamente sulla stessa casualità (ovvero il primo messaggio è sempre lo stesso) facendo domande differenti ad ogni esecuzione.

Questa tecnica viene detta rewinding.

::: Concetti Introduttivi di ZKP (22/38)



In un ZKP, si ha non solo che il prover convince il verifier della correttezza del claim, ma anche che conosce il valore x , senza comunicarlo al verifier.

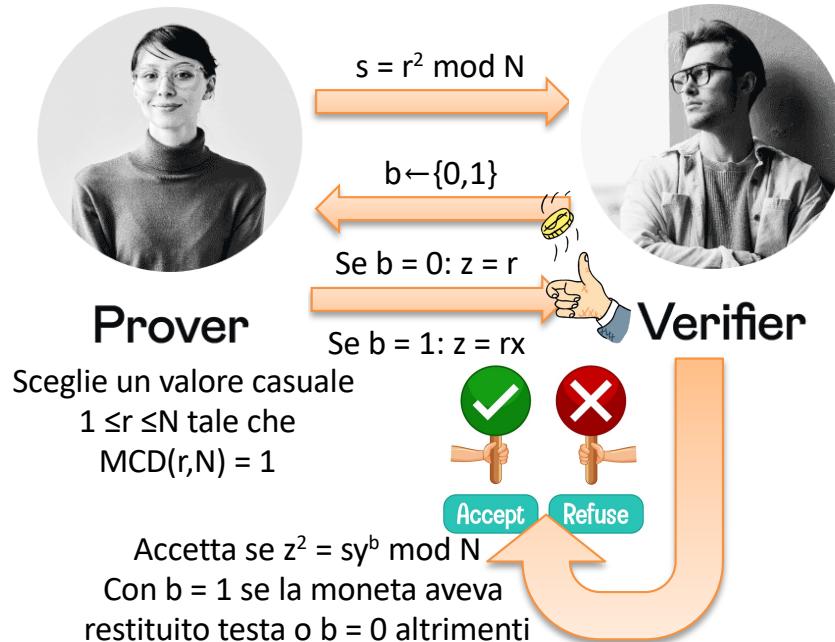
Si consideri $L_R = \{x : \exists w \text{ tale che } R(xw) = \text{accept}\}$ per la relazione R a tempo polinomiale.

(P,V) è una estrazione che $\forall x \in L_R$ accettare, con casualità (ϵ) domande differenti ad ogni richiesta.

Questo è impiegato spesso in zero-knowledge, ed è una tecnica spesso impiegata per provare che un protocollo è zk proof of knowledge: se è possibile «riavvolgere» (rewind) qualcosa molteplici volte, ed è eventualmente estrarre qualcosa da queste multiple ripetizioni, significa che era noto fin dall'inizio.

Questa tecnica viene detta **rewinding**.

::: Concetti Introduttivi di ZKP (22/38)



In un ZKP, si ha non solo che il prover convince il verifier della correttezza del claim, ma anche che conosce il valore x , senza comunicarlo al verifier.

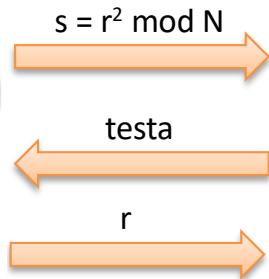
Si consideri $L_R = \{x : \exists w \text{ tale che } R(xw) = \text{accept}\}$ per la relazione R a tempo polinomiale.

(P,V) è una proof of knowledge (POK) per L_R se esiste un algoritmo di estrazione (della conoscenza) E probabilistico a tempo polinomiale tale che $\forall x \in L_R$ in tempo polinomiale $E^P(x)$ ritorna w tale che $V(x, w) = \text{accept}$, con $E^P(x)$ che indica che E esegue P ripetutamente sulla stessa casualità (ovvero il primo messaggio è sempre lo stesso) facendo domande differenti ad ogni esecuzione. Se $\text{Prob}[(P,V)(x) = \text{accetta}] > \alpha$, allora $E^P(x)$ esegue in tempo $\text{poly}(|x|, 1/\alpha)$.

::: Concetti Introduttivi di ZKP (23/38)



Prover



Estrattore

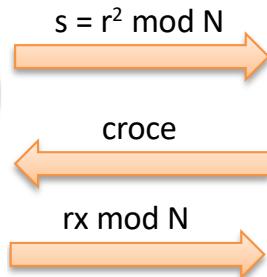
1. Esegue il prover e riceve s
2. Scrive il messaggio del verifier con testa ed ottiene r che memorizza

Ecco un esempio di estrazione.

::: Concetti Introduttivi di ZKP (23/38)



Prover



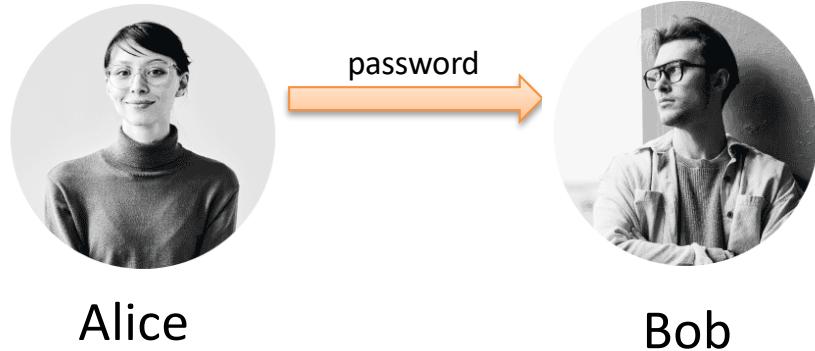
Estrattore

1. Esegue il prover e riceve s
2. Scrive il messaggio del verifier con testa ed ottiene r che memorizza
3. Riavvolge (rewind) e per il secondo run fissa il messaggio del verifier a croce e riceve rx
4. Restituisce $rx/r = x \text{ mod } N$

Ecco un esempio di estrazione.

L'estrattore ha eseguito due interazioni: una con testa e una con croce, ed ha ottenuto due risposte. Infine, l'estrattore ottiene la radice quadrata di $y \text{ mod } N$. Pertanto, esiste un estrattore e in base alla definizione, si è dimostrato che il prover conosce x.

::: Concetti Introduttivi di ZKP (24/38)



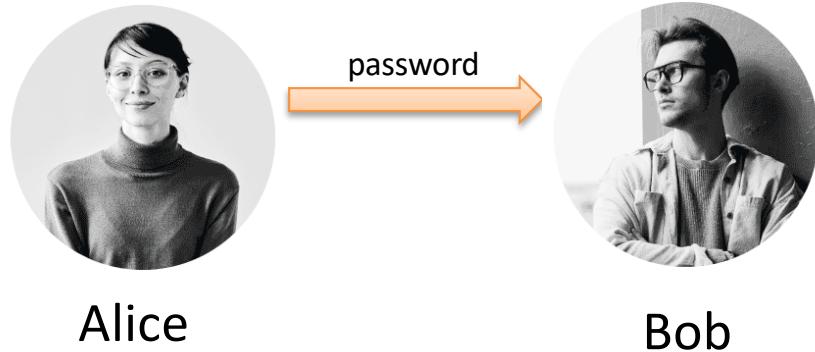
ZKP ha molte applicazioni intressanti, tra cui quella proposta da Fiat-Shamir: Alice vuole provare a Bob mediante una comunicazione (con password) su Internet che è Alice; ovviamente i messaggi possono essere intercettati.

Inoltre, Bob ha necessità di memorizzare le password al fine di poter riconoscere Alice quando presenterà una password. Per evitare che chiunque violi Bob abbia accesso a queste password, una soluzione sarebbe cifrarle, ma non lo si vuole fare per evitare di pagare un overhead in fase di verifica, oppure perché attacchi del compleanno o altri esporrebbero comunque le password memorizzate.



È possibile che Alice convinca Bob della sua identità in una maniera diversa? È possibile evitare che Bob convinca qualcun altro che lui sia Alice?

::: Concetti Introduttivi di ZKP (24/38)



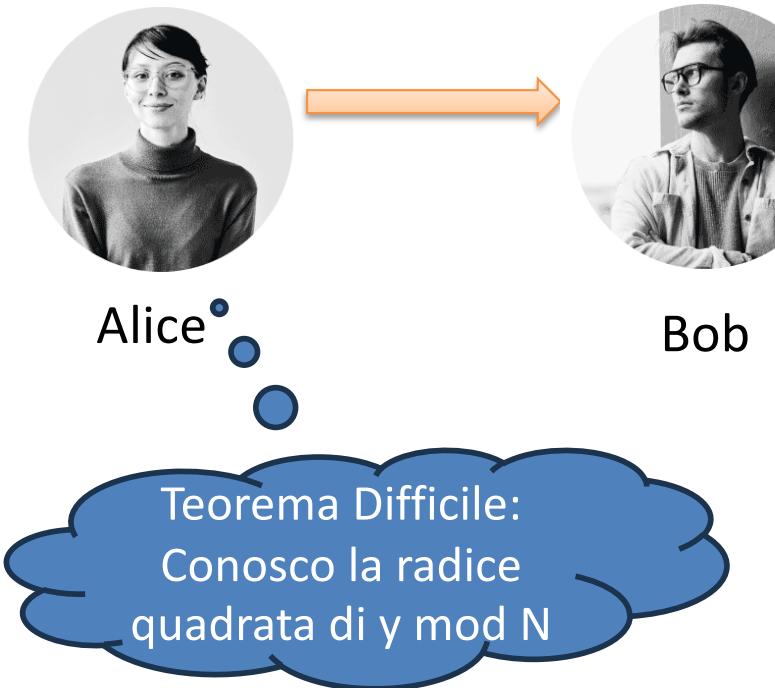
ZKP ha molte applicazioni intressanti, tra cui quella proposta da Fiat-Shamir: Alice vuole provare a Bob mediante una comunicazione (con password) su Internet che è Alice; ovviamente i messaggi possono essere intercettati.

Inoltre, Bob ha necessità di memorizzare le password al fine di poter riconoscere Alice quando presenterà una password. Per evitare che chiunque violi Bob abbia accesso a queste password, una soluzione sarebbe cifrarle, ma non lo si vuole fare per evitare di pagare un overhead in fase di verifica, oppure perché attacchi del compleanno o altri esporrebbero comunque le password memorizzate.



È possibile impiegare una zero knowledge proof.

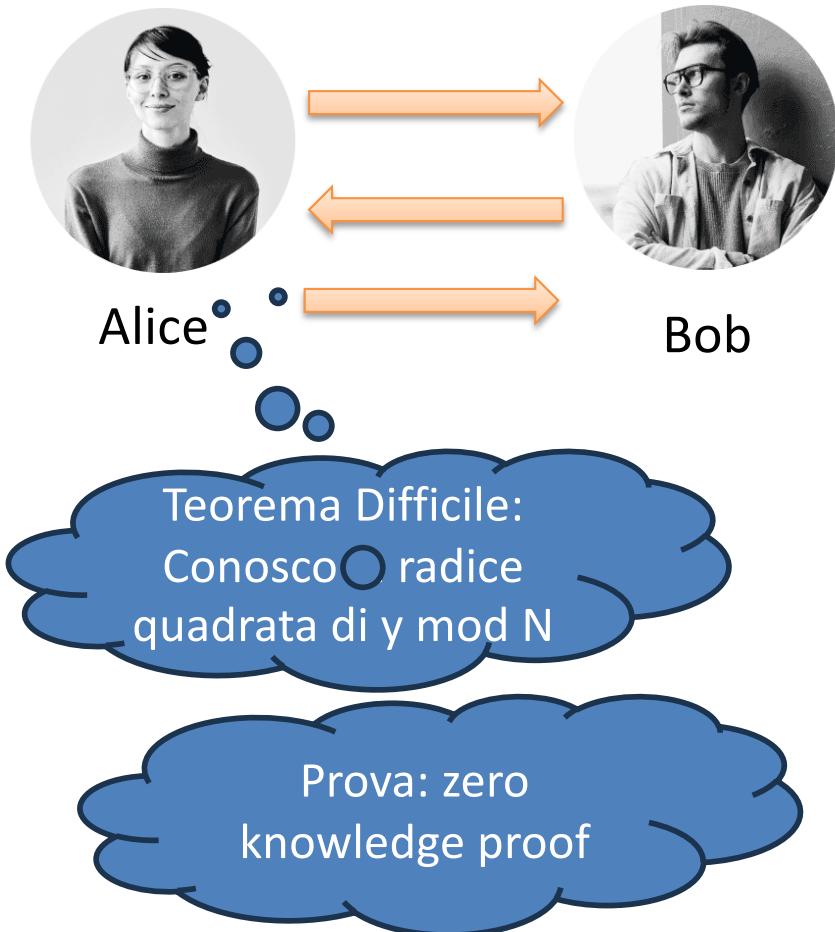
::: Concetti Introduttivi di ZKP (25/38)



La password ora è data dalla conoscenza della prova di un teorema difficile. Pertanto, chiunque conosce la soluzione, questo è Alice.

Come convincere qualcuno di essere Alice senza che questo qualcuno abbia la conoscenza che ha Alice?

::: Concetti Introduttivi di ZKP (25/38)



La password ora è data dalla conoscenza della prova di un teorema difficile. Pertanto, chiunque conosce la soluzione, questo è Alice.

Come convincere qualcuno di essere Alice senza che questo qualcuno abbia la conoscenza che ha Alice?

Alice e Bob inizieranno una serie di interazioni, alla cui conclusione Bob otterrà una zero knowledge proof della conoscenza di Alice, e tale prova identificherà Alice.

Bob potrà verificare che Alice conosce la soluzione al teorema difficile in questione senza che Bob acquisisca una soluzione al teorema stesso.

::: Concetti Introduttivi di ZKP (26/38)



Tutti i linguaggi NP hanno una prova interattiva a conoscenza zero? Bisogna analizzare ognuno dei possibili linguaggi NP e dimostrare l'esistenza di una prova a conoscenza zero?

::: Concetti Introduttivi di ZKP (26/38)



Tutti i linguaggi NP hanno una prova interattiva a conoscenza zero? Bisogna analizzare ognuno dei possibili linguaggi NP e dimostrare l'esistenza di una prova a conoscenza zero?



TEOREMA: Se una funzione one-way esiste, allora ogni linguaggio in NP ha una prova interattiva computazionale a conoscenza zero, non una perfetta.

Nell'esempio di un linguaggio sul residuo quadratico modulo N con una prova a conoscenza zero perfetta, non si è imposta alcuna condizione.

Adesso, il teorema ci impone come condizione l'esistenza di una funzione one-way. Questa è la più semplice condizione (o assunzione) in crittografia.

::: Concetti Introduttivi di ZKP (26/38)



Tutti i linguaggi NP hanno una prova interattiva a conoscenza zero? Bisogna analizzare ognuno dei possibili linguaggi NP e dimostrare l'esistenza di una prova a conoscenza zero?



TEOREMA: Se una funzione one-way esiste, allora ogni linguaggio in NP ha una prova interattiva computazionale a conoscenza zero, non una perfetta.



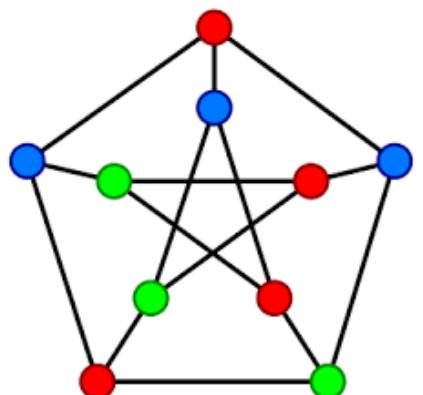
1. Un problema NP-completo ha una prova interattiva a conoscenza zero. Questo è sufficiente per avere che ogni linguaggio NP ha una prova interattiva a conoscenza zero computazionale.

::: Concetti Introduttivi di ZKP (27/38)

Un problema L è NP-completo se i seguenti enunciati sono veri:

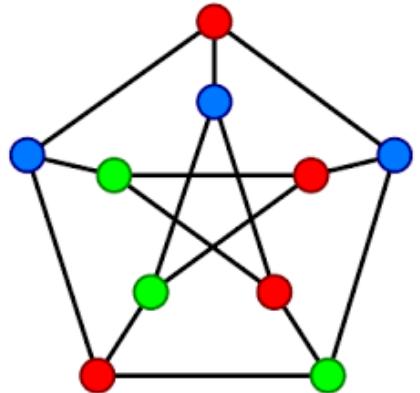
1. L è in NP;
2. Per ogni problema L' in NP esiste una riduzione polinomiale di L' in L , ovvero un algoritmo deterministico eseguibile in tempo polinomiale che trasforma istanze $I' \in L'$ in istanze $I \in L$, così che la risposta a I è sì se e solo se la risposta a I' è sì.

Nessuno ancora è stato capace di provare se i problemi NP-completi siano infatti risolvibili in tempo polinomiale. Se un linguaggio è NP-completo, ogni stringa x può essere ridotta.



Consideriamo il linguaggio NP-completo della K -colorazione per grafi non orientati (con $K \geq 3$), dove bisogna risolvere un'assegnazione di colori ai nodi del grafo in modo che non vi siano due vertici adiacenti dello stesso colore e che vengano utilizzati al massimo K colori per completare il colore del grafo.

::: Concetti Introduttivi di ZKP (28/38)



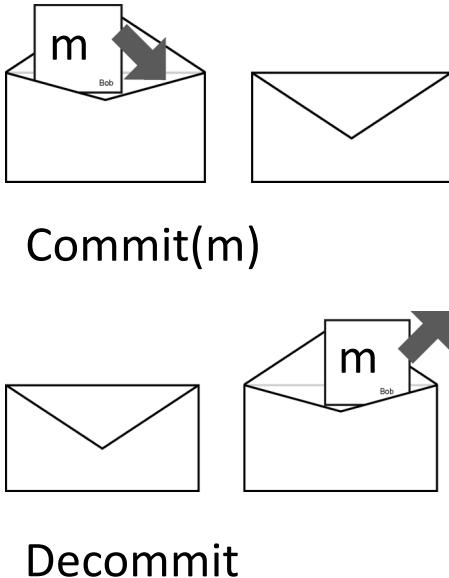
Ogni istanza x sarà ridotta a un grafo G_x tale che se x è nel linguaggio, allora G_x è 3-colorabile. Se x non è nel linguaggio allora il grafo G_x non è 3-colorabile.

Se sussiste un modo di avere una prova interattiva a conoscenza zero che dimostri che G_x è 3-colorabile, allora si ha la prova interattiva a conoscenza zero che x appartiene al linguaggio, per ogni linguaggio che è NP.

Pertanto, si è scelto di risolvere la dimostrazione di esistenza ZKP interattiva per il problema della K-colorazione, così da derivare l'esistenza per ogni linguaggio che sia NP.

Per questo scopo bisogna aggiungere un elemento fondamentale, che impiega una funzione one-way per costruire una primitiva crittografica, chiamata schema di impegno (o commitment scheme).

::: Concetti Introduttivi di ZKP (28/38)

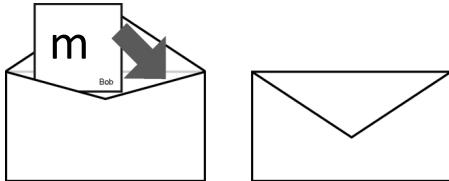


Uno schema di impegno si compone di due protocolli:

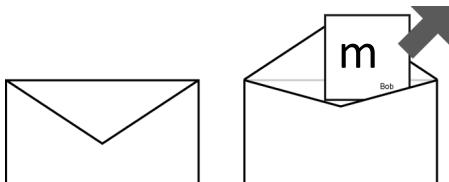
- Uno di commit, che prende in input una m , che può essere un bit che ha come valore 0 o 1, e lo mette metaforicamente in una busta e la sigilla. Il protocollo si compone di un sender, che effettua il commit, e di un receiver che riceve la busta, e non può sapere se contiene un bit 0 o 1.
- Uno di decommit, dove un sender può aprire la busta e mostrare al receiver se al suo interno c'è uno 0 o un 1.

Questi due protocolli devono soddisfare le proprietà di hiding (occultamento), durante il commit non si può conoscere il valore del bit con probabilità superiore a $\frac{1}{2}$, e binding (vincolo), ovvero una volta aperta la busta se trovo 1 non è possibile che precedentemente m era pari a 0. Quanto si è impegnato, i ha il vincolo di rivelarlo.

::: Concetti Introduttivi di ZKP (29/38)



Commit(m)



Decommit

- Hiding: \forall receiver R^* , dopo commit $\forall b, b' \in \{0,1\}$, $\{\text{View}\{\text{Sender}(b), R^*\}(1^k)\} \approx_C \{\text{View}\{\text{Sender}(b'), R^*\}(1^k)\}$ [k è un parametro di sicurezza]

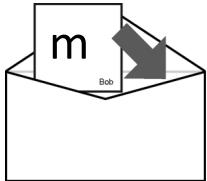
La view del receiver dopo il commit a un bit b o quella dopo il commit ad un bit b' sono computazionalmente indistinguibili.

- Binding: \forall sender S^* , al decommit Prob[Receiver deve accettare due diversi b e b'] < negl(k)

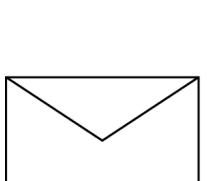
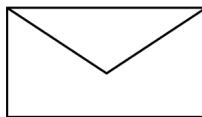
Non è possibile un decommit per due differenti valori, ovvero b e b' .

L'esempio più semplice impiega la cifratura: il commit può essere realizzato con il sender che essenzialmente cifra il bit. Se si vuole fare il commit di b , semplicemente ci cifra quel bit. Per il decommit, il receiver utilizza la chiave o la sorgente di randomness per decifrare e riottenere il bit di cui si era fatto il commit. È importante che sia uno schema randomizzato per nascondere molto fortemente il valore m .

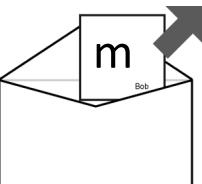
::: Concetti Introduttivi di ZKP (29/38)



Commit(m)



Decommit



- Hiding: \forall receiver R^* , dopo commit $\forall b, b' \in \{0,1\}$, $\{\text{View}\{\text{Sender}(b), R^*\}(1^k)\} \approx_C \{\text{View}\{\text{Sender}(b'), R^*\}(1^k)\}$ [k è un parametro di sicurezza]

La view del receiver dopo il commit a un bit b o quella dopo il commit ad un bit b' sono computazionalmente indistinguibili.

- Binding: \forall sender S^* , al decommit Prob[Receiver deve accettare due diversi b e b'] < negl(k)

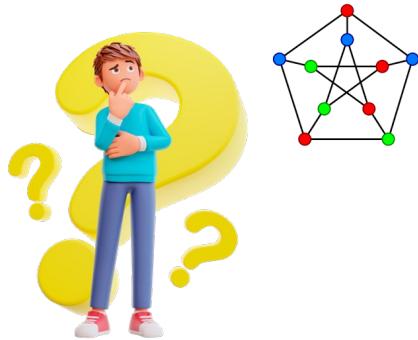
Non è possibile un decommit per due differenti valori, ovvero b e b' .

L'esempio più semplice impiega la cifratura: il commit può essere realizzato con:

commit di Bisogna impiegare la cifratura probabilistica: un algoritmo in grado di applicare la casualità a un meccanismo di crittografia così che, per ogni input di dati, ottenere un output diverso per ogni interazione effettuata.

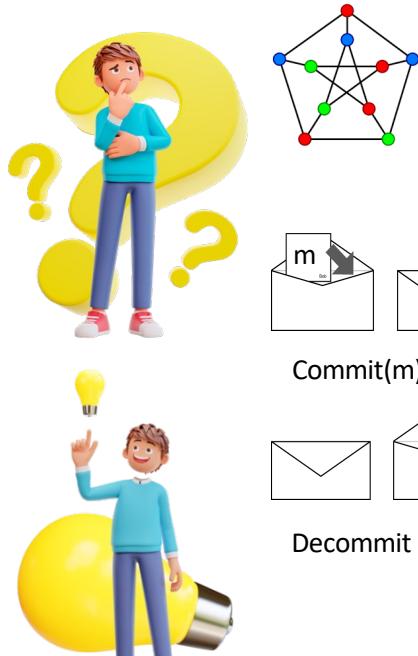
receiver utilizzando la stessa chiave di randomness per decifrare e riottenere il bit di È importante che sia uno schema randomizzato per nascondere molto fortemente il valore m .

::: Concetti Introduttivi di ZKP (30/38)



Come dimostrare che un grafo è 3-colorabile senza fornire nei fatti la soluzione ma avvalendosi di una prova computazionale a zero conoscenza?

::: Concetti Introduttivi di ZKP (30/38)



Come dimostrare che un grafo è 3-colorabile senza fornire nei fatti la soluzione ma avvalendosi di una prova computazionale a zero conoscenza?

Partiamo dal fatto che il prover ha piena conoscenza, e come il verifier, dispone del grafo, ma in aggiunta ha anche la colorazione, soluzione del problema, dove ogni vertice ha un colore, ed ogni colore corrisponde a un numero.

Conoscenza comune $G = (V, E)$, il prover ha $\pi: V \rightarrow \{0, 1, 2\}$, dove ogni numero è associato a un colore.

1. Il prover prende una permutazione a caso σ dei tre colori sui numeri associati (sul totale delle 6 possibili permutazioni), e ricolora i vertici del grafo con $\phi(v) := \sigma(\pi(v))$. Il grafo è ancora 3-colorato e impegna ognuno di questi colori eseguendo il protocollo di commit per ogni vertice del grafo ($\text{Commit}(\phi(v))$).

::: Concetti Introduttivi di ZKP (31/38)

2. Il verifier riceve i commitments ma non sa le assegnazioni nel grafo. Per verificare se l'assegnazione non risolve il problema, bisogna trovare un lato i cui estremi hanno lo stesso colore. L'assegnazione è accettata come una soluzione se non esiste un siffatto lato. Pertanto, il verifier seleziona a caso un lato $e=(a,b)$, e lo manda al prover per effettuare il decommit dei suoi estremi.
3. Il prover apre la busta per a e b agli estremi del lato selezionato dal verifier e manca i colori assegnati ai due vertici, $\phi(a)$ e $\phi(b)$.
4. Il verifier confronta i due colori, se non sono differenti ($\phi(a) = \phi(b)$) allora rigetta la prova, altrimenti ripete lo step 1.
5. Il tutto si ripete k volte, e se si arriva alla fine senza mai rigettare allora la prova viene accettata dal verifier.

In una iterazione, il verifier apprende i colori degli estremi di un lato, ma ogni ripetizione si ha una permutazione random della colorazione, quindi non si apprende mai la soluzione del problema.

::: Concetti Introduttivi di ZKP (32/38)

- Completeness: se il grafo è 3-colorato, un prover onesto può sempre convincere il verifier di accettare. Infatti, indipendentemente da quale lato viene richiesto, il verifier otterrà sempre evidenze per non rigettare la prova ad ognuna delle k iterazioni.
- Soundness: se il grafo non è 3-colorato, ci sarà sempre almeno un lato che non è colorato propriamente. La probabilità che il verifier chiede di un lato propriamente colorato su un grafo non soluzione è piccola (esponenzialmente piccola): $\text{Prob}[\text{Verifier accept}] < (1 - 1/|E|)^k < 1/e^{|E|}$ per $k = |E|^2$.
- Zero Knowledge: si ha un simulatore S che ha come input il grafo $G=(V,E)$ e non conosce la soluzione di colorazione:
 - Si sceglie a caso un lato $e=(a,b)$ del grafo G e si scelgono a caso i colori $\phi(a)$ e $\phi(b)$ in $\{0,1,2\}$ tale che $\phi(a) \neq \phi(b)$ e per gli altri vertici si assegna come colore $\phi(v) = 2$, $\forall v \neq a, b$. La view simulata si compone di $\text{Commit}(\phi(v))$, $e=(a,b)$, $\text{decommit } \phi(a)$ e $\phi(b)$.

::: Concetti Introduttivi di ZKP (33/38)

- La view simulata si compone delle prove di $\text{Commit}(\phi(v))$, che non contengono alcuna conoscenza a un distinguisher che è computazionalmente vincolato (sono una serie di informazioni cifrate), e i due colori dei vertici di un lato. Questa vista è computazionalmente indistinguibile da quanto si ha durante una vera interazione tra prover e verifier, e ciò dipende dalla proprietà di hiding del protocollo di commitment.
- Questa dimostrazione vale per un verifier onesto, ma nel caso di uno generico (onesto o meno) allora è diverso e più difficile:
 - Dato un verifier V^* per ogni iterazione i da 1 a $|E|^2$, ci sceglie un lato (a,b) a caso e si generano i commitments per i colori come nel caso del verifier onesto visto precedentemente.
 - Si esegue V^* sui commitments e si ottiene la sfida (a^*,b^*) . Se $(a^*,b^*) = (a,b)$, si ritorna l'output come nel caso del verifier onesto altrimenti l'iterazione fallisce. Se tutte le iterazioni falliscono, allora si ha come output \perp .
 - Se lo schema di commitment gode delle proprietà di Hiding e Binding, allora $\forall G, \pi(\text{una vera colorazione}): \text{prob}[\text{output } \perp] = \text{neg}(|E|)$ e se l'output non è \perp , allora $\text{simulated-view} \approx_C \text{real-view}$.

::: Concetti Introduttivi di ZKP (34/38)

Dato che tutti i linguaggi NP hanno una prova a conoscenza zero interattiva mediante questo protocollo di colorazione, ci sono tanti esempi quanti i linguaggi NP.

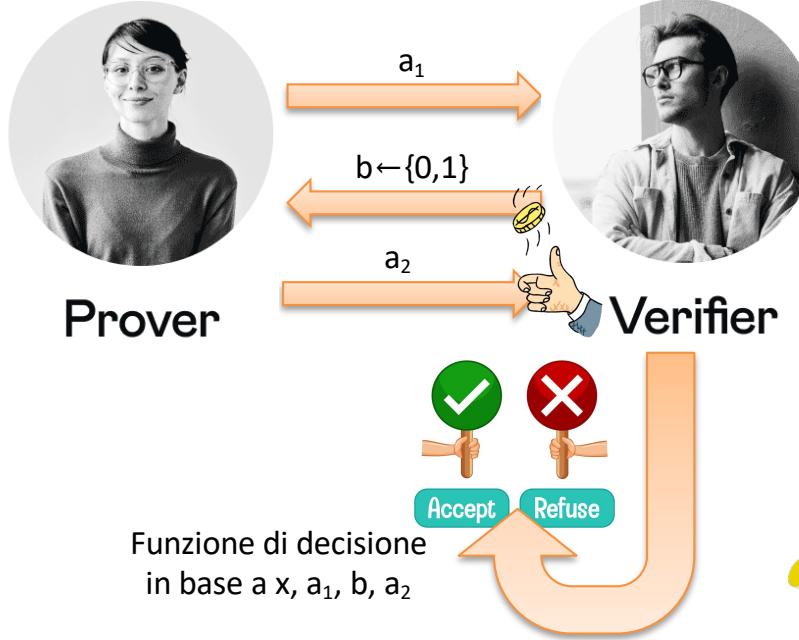
- Dimostrare che qualsiasi formula booleana soddisfacibile ha una soluzione può essere fatto a conoscenza zero senza concretamente fornire la soluzione.
 - Dato un input cifrato $E(x)$ e un programma PROG tale che $y = \text{PROG}(x)$, è possibile dimostrare a un verifier con prova a conoscenza zero che se si esegue PROG (dato cifrato al verifier) su un input x (di cui il verifier conosce il cifrato) si ottiene y . Questo è possibile senza rilevare l'input o PROG.
 - Questo risultato è importante perché è possibile provare proprietà di m senza rilevarlo ma mostrando solo il cui cifrato o il digest ottenibile dall'applicazione di una funzione di hash su m , oppure provare le relazioni tra m_1 e m_2 (come la loro uguaglianza o disuguaglianza o un qualunque operatore tra loro) senza mai rilevare nessuno ma solo i commit degli stessi.

::: Concetti Introduttivi di ZKP (35/38)

- Si può promuovere un comportamento onesto in un protocollo crittografico senza rilevare alcuna conoscenza. I partecipanti al protocollo inviano insieme al prossimo messaggio del protocollo una prova ZK interattiva per dimostrare che è il messaggio che il partecipante onesto deve mandare in accordo al protocollo, che in un contesto controllato è dimostrato essere safe e sicuro. In altre parole, il prossimo messaggio è quello che il protocollo impone sulla base della storia dei messaggi passati inviati congiuntamente a una sorgente di casualità r (e forse altri input interni segreti).
- Questo protocollo esegue il commit della sorgente di casualità r (e anche degli altri segreti interni) quando il protocollo inizia e per ogni volta bisogna mandare un messaggio, allegando al messaggio una prova interattiva ZK che quel messaggio è quello giusto.
- Questo è possibile perché è uno statement NP: esiste un fattore di causalità tale che il prossimo messaggio è quello specificato dal protocollo in base alla storia dei messaggi passati in r .

::: Concetti Introduttivi di ZKP (36/38)

Paradigma Fiat-Shamir:



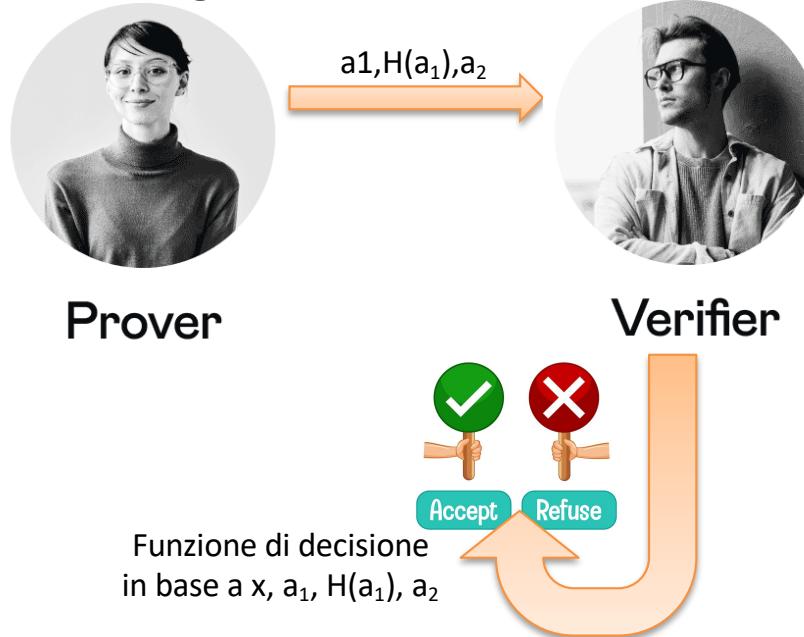
Si consideri il caso del verifier che si limita a lanciare una moneta, mandare il valore ottenuto, attendere il messaggio di risposta ed eseguire una funzione decisionale, e un prover che invia messaggi, è possibile non avere interazione nella prova ZK.



Come è possibile avere una prova ZK senza interazione?

::: Concetti Introduttivi di ZKP (36/38)

Paradigma Fiat-Shamir:



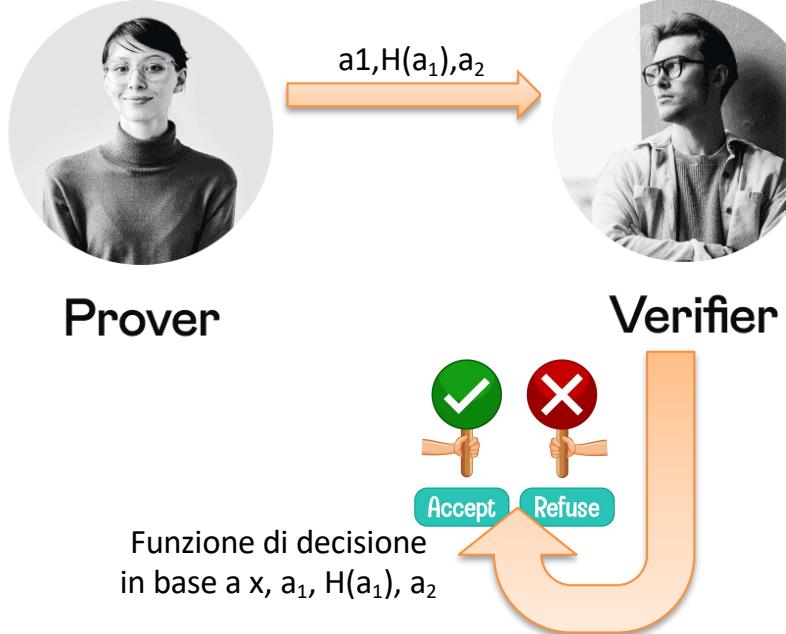
Si consideri il caso del verifier che si limita a lanciare una moneta, mandare il valore ottenuto, attendere il messaggio di risposta ed eseguire una funzione decisionale, e un prover che invia messaggi, è possibile non avere interazione nella prova ZK.

Si ipotizzi di avere disponibile una funzione di hash crittografica H che funge da oracolo random, ovvero ha in input una stringa e in output si hanno valori casuali (davvero non prevedibili).

Data una prova ZK di cui sono state provate le proprietà di completeness e soundness, il prover invece di ingaggiare una interazione, può decidere di usare H per la generazione del valore casuale che attende dal verifier. Ciò gli consente di mandare il primo messaggio del protocollo, il valore casuale generato da H e il secondo messaggio in risposta al valore casuale come se fosse stato mandato dal verifier. Dato questo input, e non una vera interazione, il verifier può eseguire la sua funzione decisionale.

::: Concetti Introduttivi di ZKP (37/38)

Paradigma Fiat-Shamir:



Questa soluzione dovrebbe soddisfare completeness e soundness come una vera interazione.

Sussistono soluzioni che impiegano questo paradigma, che è come un'euristica perché non si dispone di una tale funzione di hash crittografica. Tale paradigma è più un'idealizzazione che prende il nome di modello dell'oracolo random.

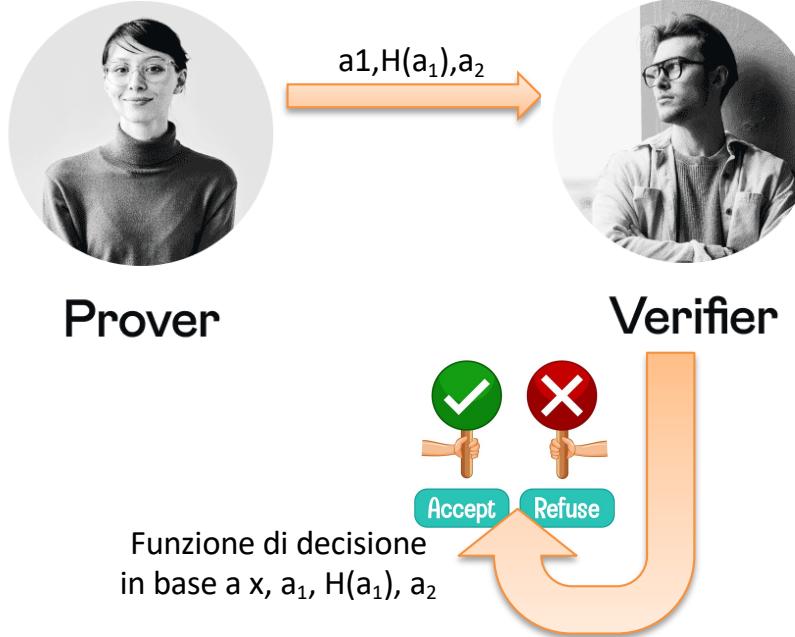
Se la funzione h è un oracolo random, allora completeness e soundness sono rispettati. Bisogna però considerare che non tutte le prove interattive a conoscenza zero possono essere rese non-interattive, ma questa soluzione può essere utile per molti protocolli.



In molti protocolli, il primo messaggio proviene dal verifier, e non dal prover, come è possibile rendere questi protocolli non-interattivi?

::: Concetti Introduttivi di ZKP (37/38)

Paradigma Fiat-Shamir:



Questa soluzione dovrebbe soddisfare completeness e soundness come una vera interazione.

Sussistono soluzioni che impiegano questo paradigma, che è come un'euristica perché non si dispone di una tale funzione di hash crittografica. Tale paradigma è più un'idealizzazione che prende il nome di modello dell'oracolo random.

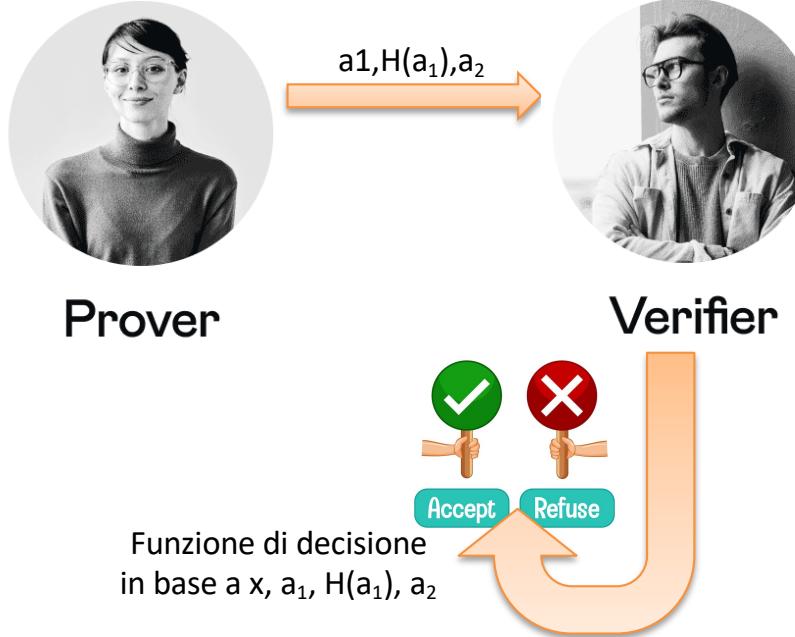
Se la funzione h è un oracolo random, allora completeness e soundness sono rispettati. Bisogna però considerare che non tutte le prove interattive a conoscenza zero possono essere rese non-interattive, ma questa soluzione può essere utile per molti protocolli.



Viene inviato il primo messaggio come una casualità scelta pubblicamente affinché tutti possano vederla, e poi applicare l'euristica Fiat-Shamir per avere prove non-interattive.

::: Concetti Introduttivi di ZKP (37/38)

Paradigma Fiat-Shamir:



Questa soluzione dovrebbe soddisfare completeness e soundness come una vera interazione.

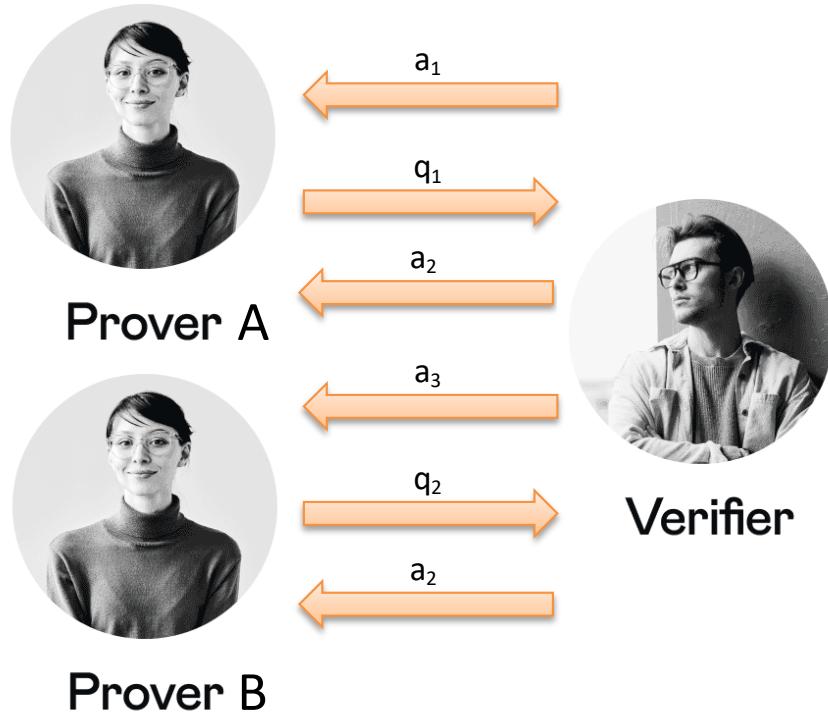
Sussistono soluzioni che impiegano questo paradigma, che è come un'euristica perché non si dispone di una tale funzione di hash crittografica. Tale paradigma è più un'idealizzazione che prende il nome di modello dell'oracolo random.

Se la funzione h è un oracolo random, allora completeness e soundness sono rispettati. Bisogna però considerare che non tutte le prove interattive a conoscenza zero possono essere rese non-interattive, ma questa soluzione può essere utile per molti protocolli.



Viene inviato il primo messaggio come una casualità scelta pubblicamente affinché tutti possano vederla, e poi applicare l'euristica Fiat-Shamir per avere prove non-interattive.

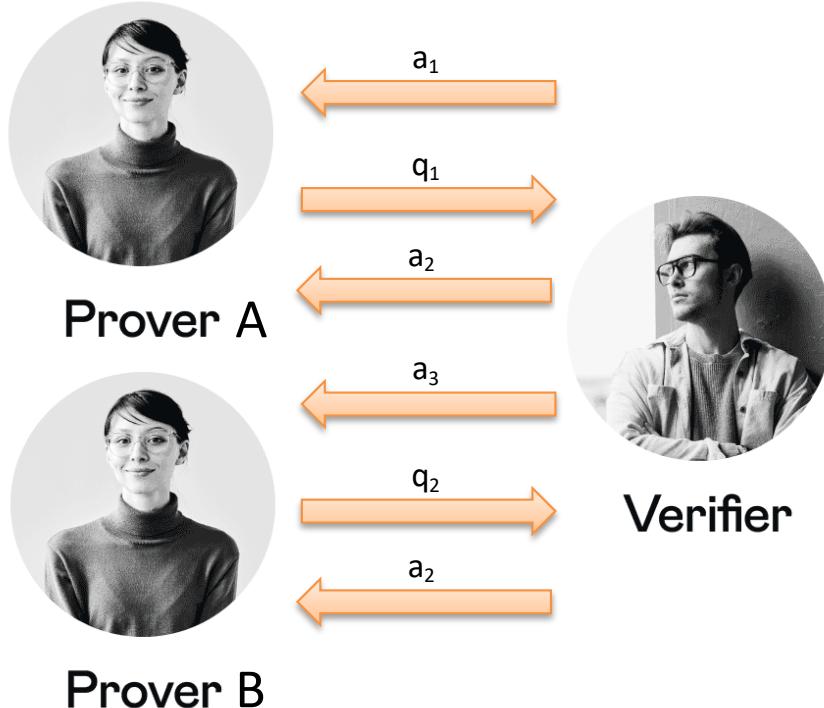
::: Concetti Introduttivi di ZKP (38/38)



Finora si è presentato schemi e soluzioni con un prover, ma cosa succede quando ve ne sono due che possono essere interrogati in maniera indipendente, senza che il primo conosca i messaggi inviati dall'altro e viceversa?



::: Concetti Introduttivi di ZKP (38/38)



Questo sistema con due prover fornisce una prova a conoscenza zero incondizionatamente, infatti è possibile identificare delle inconsistenze tra i prover che deviano per la non onesta o perché il claim è falso.

Si ottiene ZK perfetto per tutti i linguaggi NP.

Due prover possono convincere un verifier a tempo polinomiale di statement non deterministiche a tempo esponenziale.

Prover B

Teorema PCP: statement NP possono essere verificati con alta probabilità anche solo leggendo un numero contate di bit della prova, considerando due provers.

::: Schemi ZKP non interattivi (1/24)

SNARK è un acronimo che sta per Succinct Non-Interactive Argument of Knowledge, ovvero una prova succinta che una determinata affermazione (claim) è vera.

::: Schemi ZKP non interattivi (1/24)

SNARK è un acronimo che sta per Succinct Non-Interactive Argument of Knowledge, ovvero una prova **succinta** che una determinata affermazione (claim) è vera.

La prova è «breve» e «veloce» da verificare. Se l'affermazione è «conosco un messaggio m per cui $\text{SHA256}(m) = 0$ », mandare il messaggio m (lungo 1GB) per verificare questa affermazione è una prova banale ma non è succinta, perché lunga e si impiega tempo per la verifica.

::: Schemi ZKP non interattivi (1/24)

SNARK è un acronimo che sta per Succinct Non-Interactive Argument of Knowledge, ovvero una prova succinta che una determinata affermazione (claim) è vera.

zk-SNARK è uno schema di prova succinta che non rivela alcuna conoscenza rispetto all'affermazione che vuole provare.

::: Schemi ZKP non interattivi (1/24)

SNARK è un acronimo che sta per Succinct Non-Interactive Argument of Knowledge, ovvero una prova succinta che una determinata affermazione (claim) è vera.

zk-SNARK è uno schema di prova succinta che non rivela **alcuna conoscenza** rispetto all'affermazione che vuole provare.

È possibile convincere un verifier che si conosce il messaggio m ma lo si tratta come un segreto e non lo si invia al verifier per convincerlo.

::: Schemi ZKP non interattivi (1/24)

SNARK è un acronimo che sta per Succinct Non-Interactive Argument of Knowledge, ovvero una prova succinta che una determinata affermazione (claim) è vera.

zk-SNARK è uno schema di prova succinta che non rivela alcuna conoscenza rispetto all'affermazione che vuole provare.

Sussiste un grande interesse rispetto a schemi SNARK, soprattutto nelle blockchain:

- **Outsourcing computation:** una catena di livello L1 può verificare velocemente il lavoro di un servizio off-chain, come coi proof-based Rollups (zkRollup) dove la catena L1 verifica con una prova succinta che un blocco di transazioni (di notevole dimensione N) è stato processato correttamente da un servizio off-chain. Questo consente di scalare la catena L1 di un fattore N.
- **Bridging tra blockchains (zkBridge)** che consente di trasferire asset da una catena all'altra, dove la catena di origine blocca alcuni asset, da impiegare sulla catena di destinazione. La catena di destinazione deve essere convinta che quella di origine ha effettivamente bloccato quegli asset (per evitare casi di double spending o affini). Questo è possibile provando alla catena di destinazione che il protocollo di consenso sulla catena di origine ha deciso di bloccare gli asset, questo lo si fa generando una prova SNARK dalla cena di origine a quella di destinazione.

::: Schemi ZKP non interattivi (1/24)

SNARK è un acronimo che sta per Succinct Non-Interactive Argument of Knowledge,

In queste applicazioni è critico che la prova non sia interattiva, perché la prova deve essere verificata da un gran numero di validatori blockchain, e per questo non possiamo impiegare il prover in un'interazione con un tale numero di verifier per convincerli che l'affermazione è vera.

Sussiste un grande interesse rispetto a schemi SNARK, soprattutto nelle blockchain:

- **Outsourcing computation**: una catena di livello L1 può verificare velocemente il lavoro di un servizio off-chain, come coi proof-based Rollups (zkRollup) dove la catena L1 verifica con una prova succinta che un blocco di transazioni (di notevole dimensione N) è stato processato correttamente da un servizio off-chain. Questo consente di scalare la catena L1 di un fattore N.
- **Bridging tra blockchains** (zkBridge) che consente di trasferire asset da una catena all'altra, dove la catena di origine blocca alcuni asset, da impiegare sulla catena di destinazione. La catena di destinazione deve essere convinta che quella di origine ha effettivamente bloccato quegli asset (per evitare casi di double spending o affini). Questo è possibile provando alla catena di destinazione che il protocollo di consenso sulla catena di origine ha deciso di bloccare gli asset, questo lo si fa generando una prova SNARK dalla cena di origine a quella di destinazione.

::: Schemi ZKP non interattivi (2/24)

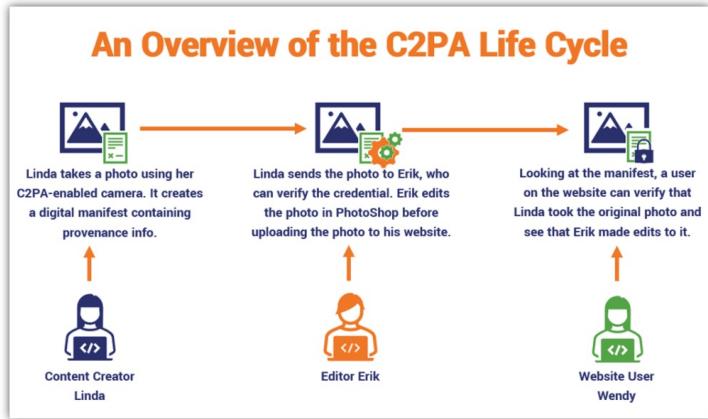
Altre applicazioni nel contesto blockchain richiedono privacy e per questo è necessario impiegare zk-SNARK:

- Processare delle transazioni private (ovvero quando memorizzate sulla catena, le informazioni sono cifrate e quindi rese non disponibili per un'ispezione pubblica) su una blockchain pubblica. Siccome i dati della transazione sono privati, si associa una prova zk che convince della validità della transazione, che è stata propriamente firmata e che nessuna valuta viene creata o persa.
- Compliance rispetto a determinate regole di governance quando si ha a che fare con transazioni private, e il creatore della transazione vuole provare che tale transazione rispetta delle regole bancarie associando una prova zk. Un altro esempio è quando si vuole provare che si dispongono degli asset necessari alla precondizione di una transazione.

Esistono anche applicazioni non nel contesto delle blockchain :

- Nelle notizie pubblicate nel digitale è possibile che il post parli di un evento mentre l'immagine associata provenga da un altro evento. Quindi sussiste il problema di identificare immagini forvianti nei post/news pubblicate nel web. Sussiste un recente standard C2PA il cui scopo è di fornire l'autentica provenienza delle immagini.

::: Schemi ZKP non interattivi (3/24)



L'idea è quella di integrare in ogni fotocamera una chiave segreta inserita dal costruttore. Tale chiave non può essere estratta dalla fotocamera. Ogni volta che una fotocamera conforme a C2PA fa una foto, la firma usando quella chiave, congiuntamente a certi metadati (posizione, e). Tale «manifesto» è integrato nel file raw che contiene la foto ed è generato dalla fotocamera.

Quando l'immagine viene usata per un post/news, il lettore può verificare il manifesto e verificare se l'immagine è stata fatta nel luogo/tempo riferito nell'articolo.

Un problema di questo standard è il post-processing: la fotocamera genera immagini ad alta risoluzione ma sul web si preferiscono immagini ad una risoluzione/qualità minore per risparmiare banda. Quindi, le immagini vengono ridimensionate, tagliate, e/o modificate prima che vengano pubblicate. Tale immagine processata è difforme dall'originale e quindi la verifica fallisce (la firma deve essere verificata rispetto all'immagine originale).

Invece di integrare una firma, viene usata una prova zk-SNARK dal programma di editazione, data l'immagine dopo la processazione e la lista di operazioni applicate.

::: Schemi ZKP non interattivi (4/24)

La prova zk-SNARK viene costruita come segue:

- Si conosce la coppia $(\text{Orig}, \text{Sig})$ dove Sig è una firma C2PA valida per l'immagine Origin , che Photo è il risultato dell'applicazione delle operazioni nella lista Ops a Orig , e i metadati dell'immagine originale e di quella processata sono uguali.
- Il verifier (ovvero il device usato dal lettore) verifica la prova e mostra i metadati all'utente. La prova è molto breve, meno di 1KB e il tempo di verifica è breve, meno di 10 ms. Di contro, il tempo per generare la prova (da fare una tantum) per immagini di 6000x4000 pixels è di circa qualche minuto. La generazione, però, è fortemente parallelizzabile, e impiegando più macchine è possibile abbattere il tempo di generazione a qualche secondo. La prova deve essere non-interattiva siccome il fornitore del servizio non può interagire con ogni lettore per provare la provenienza delle immagini che impiega nei suoi post/news. Il fornitore genera la prova che viene inviata a tutti i lettori che in autonomia verificano la veridicità e correttezza della prova.

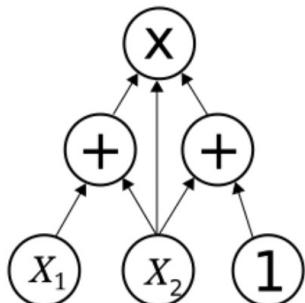
Tutte queste applicazioni (con prove su affermazioni molto grandi e complesse) sono state possibili negli ultimi 10 anni grazie alla possibilità di generare prove in un tempo lineare (quasi lineare) rispetto alla dimensione di computazione.

::: Schemi ZKP non interattivi (5/24)

Per definire SNARK bisogna fissare il modello computazionale:

- Dato un insieme finito $\mathbb{F} = \{0, \dots, p - 1\}$ per un grande numero primo $p > 2$ e una coppia di operazioni di addizione e moltiplicazione che ritorna un risultato modulo p , si ottiene un campo finito.
- Un circuito aritmetico è fondamentalmente una funzione $C: \mathbb{F}^n \rightarrow \mathbb{F}$, ovvero che ha come input elementi del campo e che produce elementi nel campo.

$$x_2(x_1 + x_2)(x_2 + 1)$$



È definibile come un Directed Acyclic Graph (DAG) dove i nodi interni sono labellati con le operazioni del campo e gli input possono essere sia delle variabili che delle costanti del campo.

Ammette anche una rappresentazione come polinomio di n variabili e come una «ricetta» per la valutazione del polinomio.

$|C|$ indica il numero di gate (o operazioni) nel circuito C .

$$|x_2(x_1 + x_2)(x_2 + 1)| = 3$$

In letteratura si trovano molti circuiti aritmetici, che possono fare qualunque cosa computabile in tempo polinomiale.

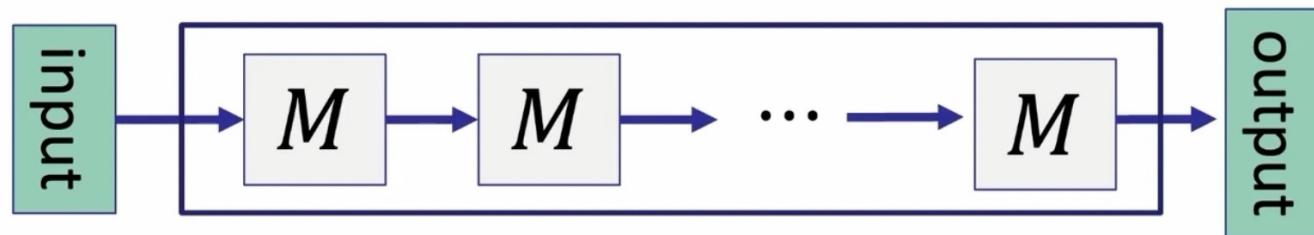
- $C_{SHA}(h, m)$ è un circuito che data una funzione di hash restituisce 0 se $SHA256(m) = h$ e un risultato non nullo altrimenti. $C_{hSHA}(h, m) = (h - SHA256(m))$. $|C_{SHA}| \approx 20K$ gate (esempio di un piccolo circuito aritmetico).

::: Schemi ZKP non interattivi (6/24)

- $C_{sig}(pk, m, \sigma)$ è un circuito che restituisce 0 se σ è una firma ECDSA valida di m rispetto a una chiave pubblica pk .

Spesso si distinguono tra due tipi di circuiti aritmetici:

- Un circuito non strutturato è uno in cui i collegamenti hanno una topologia caotica, con un numero arbitrario di gate e i collegamenti che sono stabiliti tra i gate all'occorrenza.
- Un circuito strutturato è quello progettato a layer che sono ripetuti varie volte al fine di ottenere l'output desiderato. Un layer M è spesso chiamato virtual machine e può essere visto come una specie di microprocessore.



Alcuni schemi SNARK sono generici e si applicano su entrambe le tipologie ma ci sono schemi SNARK che si applicano solo su circuiti strutturati.

::: Schemi ZKP non interattivi (7/24)

NARK sta per Non-interactive Argument of Knowledge ed è applicato a un circuito aritmetico $C(x, w) \rightarrow \mathbb{F}$ che prende in input un'affermazione pubblica in \mathbb{F}^n e un testimone segreto in \mathbb{F}^m e restituisce un risultato nel campo finito \mathbb{F} .

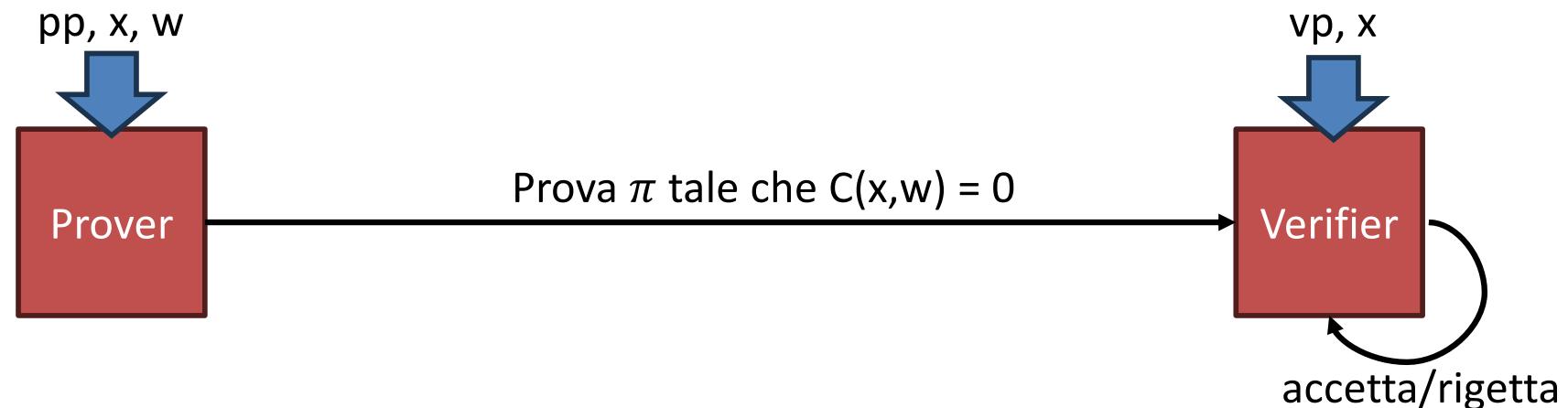
NARK funziona con un algoritmo di pre-processazione, chiamato S , che prende la descrizione del circuito come input e restituisce dei parametri pubblici chiamati pp (per il prover) e vp (per il verifier). Prover e verifier a questo punto hanno i loro input:



::: Schemi ZKP non interattivi (7/24)

NARK sta per Non-interactive Argument of Knowledge ed è applicato a un circuito aritmetico $C(x, w) \rightarrow \mathbb{F}$ che prende in input un'affermazione pubblica in \mathbb{F}^n e un testimone segreto in \mathbb{F}^m e restituisce un risultato nel campo finito \mathbb{F} .

NARK funziona con un algoritmo di pre-processazione, chiamato S , che prende la descrizione del circuito come input e restituisce dei parametri pubblici chiamati pp (per il prover) e vp (per il verifier). Prover e verifier a questo punto hanno i loro input:



Il prover genera una prova π con cui convincere il verifier che conosce un determinato input w tale che il circuito dato x e w restituisce 0. Il verifier controlla la prova e nel caso la accetta o rigetta.

::: Schemi ZKP non interattivi (8/24)

Data questa definizione possiamo dire che uno schema NARK è definibile come una tripla (S, P, V) con

- $S(C) \rightarrow (pp, vp)$ è l'algoritmo di generazione dei parametri pubblici per il prover e verifier;
- $P(pp, x, w) \rightarrow \pi$ è l'algoritmo di generazione della prova π ;
- $V(vp, x, \pi) \rightarrow accept/reject$ è l'algoritmo di verifica che restituisce accettazione o rigetto.

Tutti questi algoritmi in pratica hanno anche accesso al cosiddetto oracolo random.

Ci sono una serie di requisiti che uno schema NARK deve soddisfare:

- Completeness: se una prova è valida, deve essere accettata (è perfetta se l'accettazione avviene sempre).
$$\forall x, w: C(x, w) = 0 \Rightarrow \Pr[V(vp, x, P(pp, x, w)) = accept] = 1$$
- Knowledge sound: se l'algoritmo V accetta una prova dal prover, allora il prover «conosce» veramente un testimone w tale che $C(x, w) = 0$, ovvero esiste un Extractor E può ottenere un w valida dal prover.
- (Opzionale) Zero Knowledge: tutto quello che il verifier acquisisce (C, pp, vp, x, π) non rivela nulla di w .

::: Schemi ZKP non interattivi (9/24)

Uno schema NARK banale è che la prova π è semplicemente pari a w , così che il verifier deve solo eseguire il circuito e verificare se restituisce 0.

Questo non è quello che si vuole, ma piuttosto uno schema di pre-processazione NARK succinto, ovvero la tripla (S, P, V) con

- $S(C) \rightarrow (pp, vp)$ è l'algoritmo di generazione dei parametri pubblici per il prover e verifier;
- $P(pp, x, w) \rightarrow \pi$ genera una prova breve, ovvero $len(\pi) = \text{sublinear}(|w|)$;
- $V(vp, x, \pi) \rightarrow \text{accept/reject}$ è veloce da verificare, ovvero $time(V) = O_\lambda(|x|, \text{sublinear}(|C|))$.

Il secondo requisito non rende possibile di avere la prova π semplicemente pari a w , mentre il terzo non consente di rieseguire il circuito (il tempo di esecuzione deve essere sotto-lineare alla dimensione del circuito), ma consente di essere lineari con la dimensione di x . Un esempio di funzione sotto-lineare è la radice quadrata.

Uno SNARK è uno schema che performa meglio di un banale NARK.

Nella realtà si vuole essere più greedy, e questa caratterizzazione non va ancora bene.

::: Schemi ZKP non interattivi (10/24)

Si vuole uno schema di pre-processazione NARK fortemente succinto, ovvero la tripla (S, P, V) con

- $S(C) \rightarrow (pp, vp)$ è l'algoritmo di generazione dei parametri pubblici per il prover e verifier;
- $P(pp, x, w) \rightarrow \pi$ genera una prova breve, ovvero $\text{len}(\pi) = O_\lambda(\log |C|)$;
- $V(vp, x, \pi) \rightarrow \text{accept/reject}$ è veloce da verificare, ovvero $\text{time}(V) = O_\lambda(|x|, \log(|C|))$.

Nella pratica, gli schemi attuali hanno come dimensione di prova e tempo di verifica un valore costante indipendentemente dalla dimensione del circuito e dell'affermazione x .

È possibile notare che il verifier non ha il tempo di «leggere» il circuito C perché deve essere estremamente veloce. La fase di pre-processazione serve proprio a questo scopo. Infatti, vp rappresenta un sunto del circuito C per il verifier generato dall'algoritmo S .

SNARK è uno schema NARK (che garantisce completeness e knowledge sound) con vincoli molto stringenti sulla lunghezza della prova e sul tempo di verifica della stessa. zk-SNARK è uno SNARK che gode anche della proprietà di privacy.

::: Schemi ZKP non interattivi (11/24)

Un algoritmo di setup tipicamente è nella forma $S(C; r) \rightarrow (pp, vp)$ con r dei bit random. Sussistono tre tipi di algoritmi:

- Un setup fidato per circuiti: l'algoritmo deve essere eseguito come nuovo per ogni circuito che si vuole processare, ed è critico che i bit random r sono mantenuti segreti al prover per evitare che possa provare delle affermazioni false. Nel contesto di una blockchain, provare statement falsi causerebbe che il prover è in grado di creare valuta o di rubarla. Successivamente alla generazione dei parametri pubblici, la macchina che ha partecipato all'esecuzione di S viene distrutta così nessuno potrà conoscere i valori di r .
- Un setup fidato ma universale (aggiornabile): il segreto r è indipendente da C e l'algoritmo è diviso in due differenti procedure, $S = (S_{init}, S_{index})$, dove la prima viene eseguita una sola volta per generare dei parametri globali:

$$S_{init}(\lambda; r) \rightarrow gp$$

La macchina che partecipa a questo processo viene distrutta così che r rimane segreto. Poi, si ha un algoritmo deterministico che prende in input i parametri globali e la descrizione del circuito per le generazione dei parametri globali:

$$S_{index}(gp, C) \rightarrow (pp, vp)$$

Ognuno può verificare che i parametri pubblici sono stati generati correttamente.

::: Schemi ZKP non interattivi (11/24)

Un algoritmo di setup tipicamente è nella forma $S(C; r) \rightarrow (pp, vp)$ con r dei bit random. Sussistono tre tipi di algoritmi:

- Un setup fidato per circuiti: l'algoritmo deve essere eseguito come nuovo per ogni circuito. A tale scopo, se si esegue un setup una sola volta, è possibile generare vari r che sono parametri pubblici per vari circuiti. È una soluzione ancora fidata grazie alle affermazioni della S_{init} , ma è utilizzabile per vari circuiti senza dover rigenerare r e la macchina di esecuzione della procedura.
- Un setup fidato ma **universale** (aggiornabile): il segreto r è indipendente da C e l'algoritmo è diviso in due differenti procedure, $S = (S_{init}, S_{index})$, dove la prima viene eseguita una sola volta per generare dei parametri globali:
$$S_{init}(\lambda; r) \rightarrow gp$$

La macchina che partecipa a questo processo viene distrutta così che r rimane segreto. Poi, si ha un algoritmo deterministico che prende in input i parametri globali e la descrizione del circuito per le generazioni dei parametri globali:

$$S_{index}(gp, C) \rightarrow (pp, vp)$$

Ognuno può verificare che i parametri pubblici sono stati generati correttamente.

::: Schemi ZKP non interattivi (12/24)

- Un setup trasparente: dove non si ha nessun dato segreto per la generazione dei parametri pubblici, ma tutti possono verificare che i parametri siano stati correttamente generati.

Questa lista rappresenta anche un ordinamento dal peggiore al migliore.

Ecco una lista (limitata) degli schemi SNARK tra i più usati per circuiti $\approx 2^{20}$ gate:

	Lunghezza della prova π	Tempo di verifica	Setup	Post-quantum?
Groth'16	≈ 200 Bytes $O_\lambda(1)$	$\approx 1,5$ ms $O_\lambda(1)$	Fidato per circuito	No
Plonk/Marlin	≈ 400 Bytes $O_\lambda(1)$	≈ 3 ms $O_\lambda(1)$	Universale e fidato	No
Bulletproofs	$\approx 1,5$ KB $O_\lambda(\log C)$	≈ 3 sec $O_\lambda(C)$	Trasparente	No
STARK	≈ 100 KB $O_\lambda(\log^2 C)$	≈ 10 ms $O_\lambda(\log^2 C)$	Trasparente	Yes

Per tutti questi schemi il tempo di esecuzione del prover è quasi lineare in $|C|$.

::: Schemi ZKP non interattivi (13/24)

Cosa significa che il prover «conosce» w nella definizione di knowledge soundness? Informalmente, si può dire che il prover «conosce» w , se w può essere «estratto» da esso. Ecco una definizione più formale di questo concetto:

- La tripla (S, P, V) soddisfa la proprietà di knowledge sound (adattativamente) per un circuito C se è vero quanto segue:
 - sia dato un avversario a tempo polinomiale $A = (A_0, A_1)$ che funga da prover malizioso ovvero che provi a convincere il verifier di uno statement senza conoscenza del testimone. Questo avversario si compone di due algoritmi.
 - Dati i parametri globali generati dalla prima procedura di setup $gp \leftarrow S_{init}()$, si impiega la prima procedura dell'avversario per generare la descrizione del circuito C e lo statement x per cui l'avversario vuole fabbricare una prova, ovvero $(C, x, st) \leftarrow A_0(gp)$, st rappresenta uno stato interno dell'avversario.
 - Successivamente viene eseguita la seconda procedura di setup per la generazione dei parametri pubblici, ovvero $(pp, vp) \leftarrow S_{index}(C)$, e anche la seconda procedura dell'avversario per ottenere la prova, ovvero $\pi \leftarrow A_1(pp, x, st)$.
 - La prova π viene consegnata al verifier congiuntamente allo statement x e la probabilità di accettazione è data da $\Pr[V(vp, x, \pi) = accept] > 1/10^6$, ovvero non trascurabile. Questo significa che l'avversario $A = (A_0, A_1)$ è stato in grado di generare un circuito e uno statement x congiuntamente ad una prova che il verifier accetterà con una probabilità uno su un milione.
- Se questo è vero, allora esisterà un algoritmo di estrazione efficiente E (che impiega A) che lavora come segue:

::: Schemi ZKP non interattivi (14/24)

- Dati i parametri globali generati dalla prima procedura di setup $gp \leftarrow S_{init}()$, si impiega la prima procedura dell'avversario per generare la descrizione del circuito C e lo statement x , ovvero $(C, x, st) \leftarrow A_0(gp)$.
- Viene eseguito l'estrattore $w \leftarrow E(gp, C, x)$, che interagisce in qualche modo con A_1 , ed è capace di estrarre il testimone ovvero $\Pr[C(x, w) = 0] > \frac{1}{10^6} - \varepsilon$, con un ε trascurabile.

Esiste un paradigma molto generale per creare uno schema SNARK per circuiti generali, che si compone di due passi:

- Uno schema di commitment funzionale, che è un oggetto crittografico dove la sua sicurezza dipende da determinate assunzioni crittografiche;
- Una prova oracolo interattiva compatibile (IOP), che è un oggetto di teoria dell'informazione dove il livello di sicurezza è incondizionatamente senza alcuna assunzione.

::: Schemi ZKP non interattivi (14/24)

- Dati i parametri globali generati dalla prima procedura di setup $gp \leftarrow S_{init}()$, si impiega la prima procedura dell'avversario per generare la descrizione del circuito C e lo statement x , ovvero $(C, x, st) \leftarrow A_0(gp)$.
- Viene eseguito l'estrattore $w \leftarrow E(gp, C, x)$, che interagisce in qualche modo con A_1 , ed è capace di estrarre il testimone ovvero $\Pr[C(x, w) = 0] > \frac{1}{10^6} - \varepsilon$, con un ε trascurabile.

Esiste un paradigma molto generale per creare uno schema SNARK per circuiti generali, che si compone di due passi:

- Uno schema di **commitment** funzionale, che è un oggetto crittografico dove la sua sicurezza dipende da determinate assunzioni crittografiche;
- Una prova on-the-fly non interattiva compatibile (IOP), che è un oggetto di teoria dell'informazione il cui livello di sicurezza è incondizionatamente senza alcuna

Uno schema di commitment è composto di due algoritmi: $commit(m, r) \rightarrow com$, con r scelto a caso, e $verify(m, com, r) \rightarrow accept/reject$ che significa l'apertura del commitment com è pari a m o non. Questo schema deve soddisfare le proprietà di hiding e binding.

::: Schemi ZKP non interattivi (14/24)

- Dati i parametri globali generati dalla prima procedura di setup $gp \leftarrow S_{init}()$, si impiega la prima procedura dell'avversario per generare la descrizione del circuito C e lo statement x , ovvero $(C, x, st) \leftarrow A_0(gp)$.
- Viene eseguito l'estrattore $w \leftarrow E(gp, C, x)$, che interagisce in qualche modo con A_1 , ed è capace di estrarre il testimone ovvero $\Pr[C(x, w) = 0] > \frac{1}{10^6} - \varepsilon$, con un ε trascurabile.

Esiste un paradigma molto generale per creare uno schema SNARK per circuiti generali, che si compone di due passi:

- Uno schema di commitment funzionale, che è un oggetto crittografico dove la sua sicurezza dipende da determinate assunzioni crittografiche;
- Una prova oracolo interattiva compatibile (IOP), che è un oggetto di teoria dell'informazione dove il livello di sicurezza è incondizionatamente senza alcuna assunzione.

Esiste una costruzione standard per schemi di commitment che impiega una funzione di hash, dove

- $commit(m, r)$: $com := H(m, r)$;
- $verify(m, com, r)$: accept if $com == H(m, r)$;

e per un'adeguata funzione di hash le proprietà di hiding e biding sono rispettate.

::: Schemi ZKP non interattivi (14/24)

- Dati i parametri globali generati dalla prima procedura di setup $gp \leftarrow S_{init}()$, si impiega la prima procedura dell'avversario per generare la descrizione del circuito C e lo statement x , ovvero $(C, x, st) \leftarrow A_0(gp)$.
- Viene eseguito l'estrattore $w \leftarrow E(gp, C, x)$, che interagisce in qualche modo con A_1 , ed è capace di estrarre il testimone ovvero $\Pr[C(x, w) = 0] > \frac{1}{10^6} - \varepsilon$, con un ε trascurabile.

Esiste un paradigma molto generale per creare uno schema SNARK per circuiti generali, che si compone di due passi:

- Uno schema di commitment funzionale, che è un oggetto crittografico dove la sua sicurezza dipende da determinate assunzioni crittografiche;
- Una prova oracolo interattiva compatibile (IOP), che è un oggetto di teoria dell'informazione dove il livello di sicurezza è incondizionatamente senza alcuna assunzione.



Questo schema base non basta e serve uno schema di soddisfi delle proprietà aggiuntive. Come fare?

::: Schemi ZKP non interattivi (15/24)

Abbiamo bisogno di uno schema di commitment funzionale: data una famiglia di funzioni $\mathcal{F} = \{f: X \rightarrow Y\}$, lo schema lavora per funzioni in questo insieme.

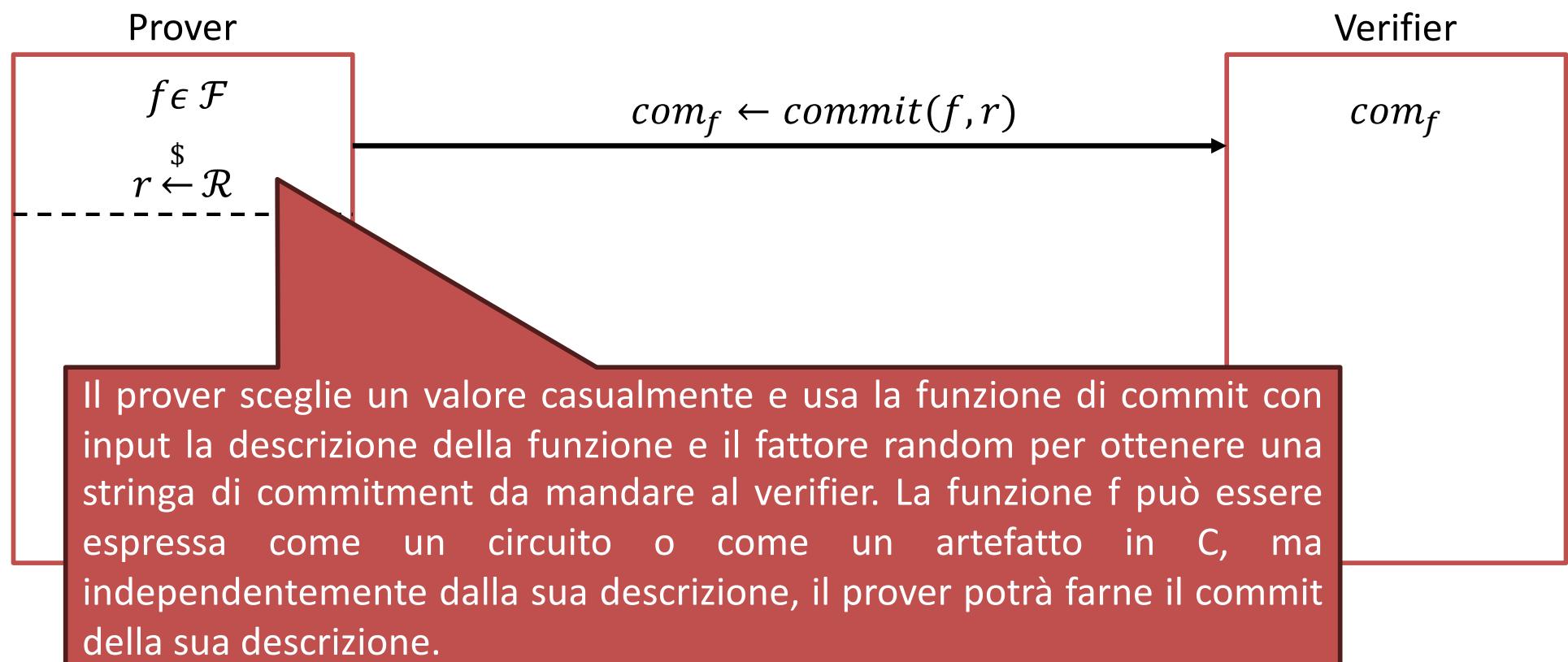
Prover

$$f \in \mathcal{F}$$

Verifier

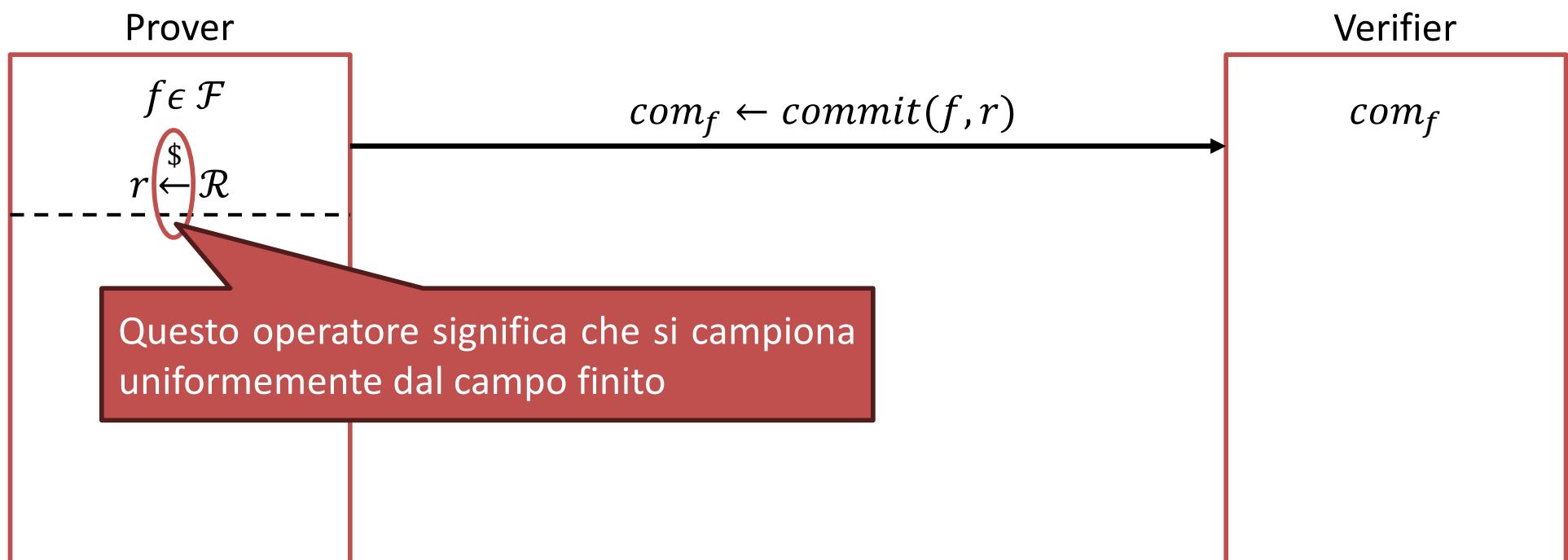
::: Schemi ZKP non interattivi (15/24)

Abbiamo bisogno di uno schema di commitment funzionale: data una famiglia di funzioni $\mathcal{F} = \{f: X \rightarrow Y\}$, lo schema lavora per funzioni in questo insieme.



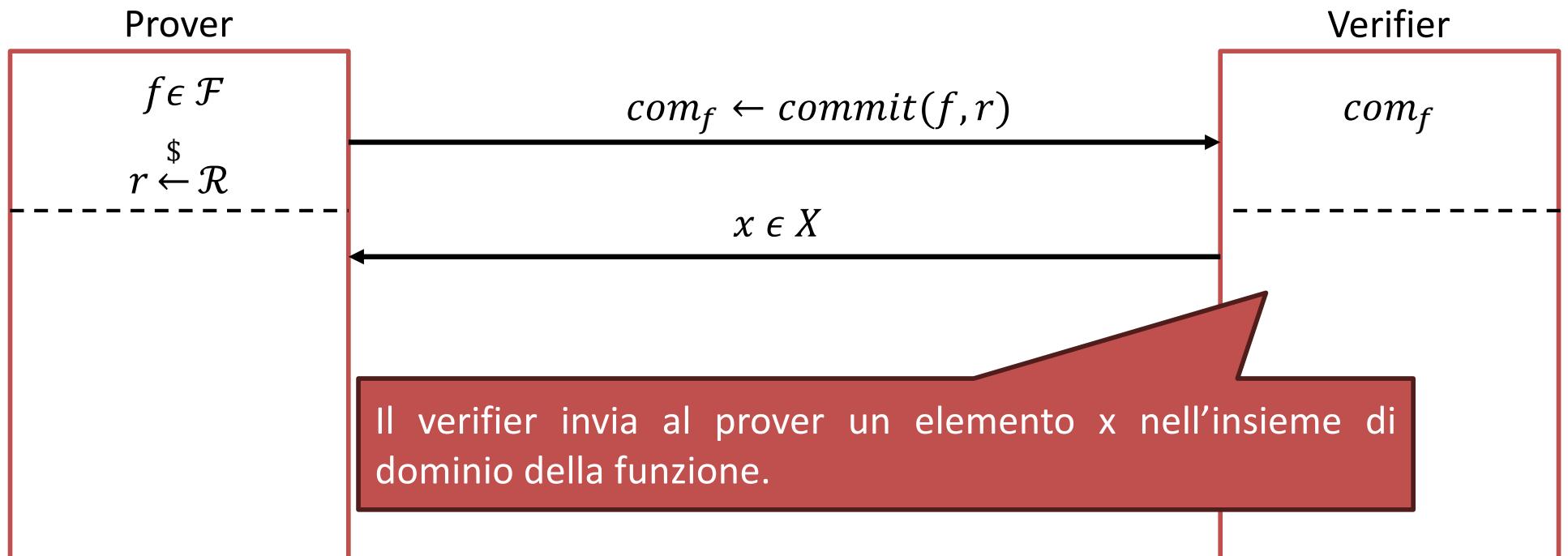
::: Schemi ZKP non interattivi (15/24)

Abbiamo bisogno di uno schema di commitment funzionale: data una famiglia di funzioni $\mathcal{F} = \{f: X \rightarrow Y\}$, lo schema lavora per funzioni in questo insieme.



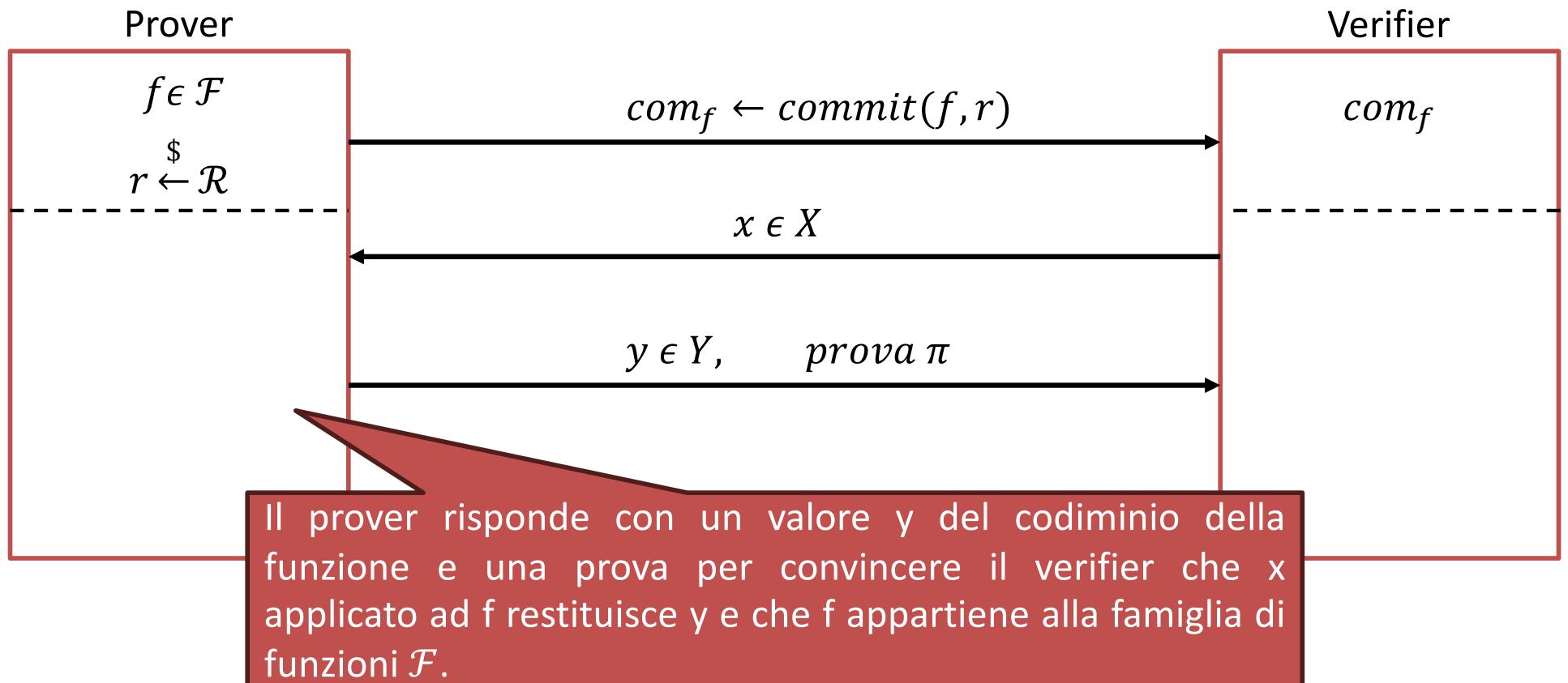
::: Schemi ZKP non interattivi (15/24)

Abbiamo bisogno di uno schema di commitment funzionale: data una famiglia di funzioni $\mathcal{F} = \{f: X \rightarrow Y\}$, lo schema lavora per funzioni in questo insieme.



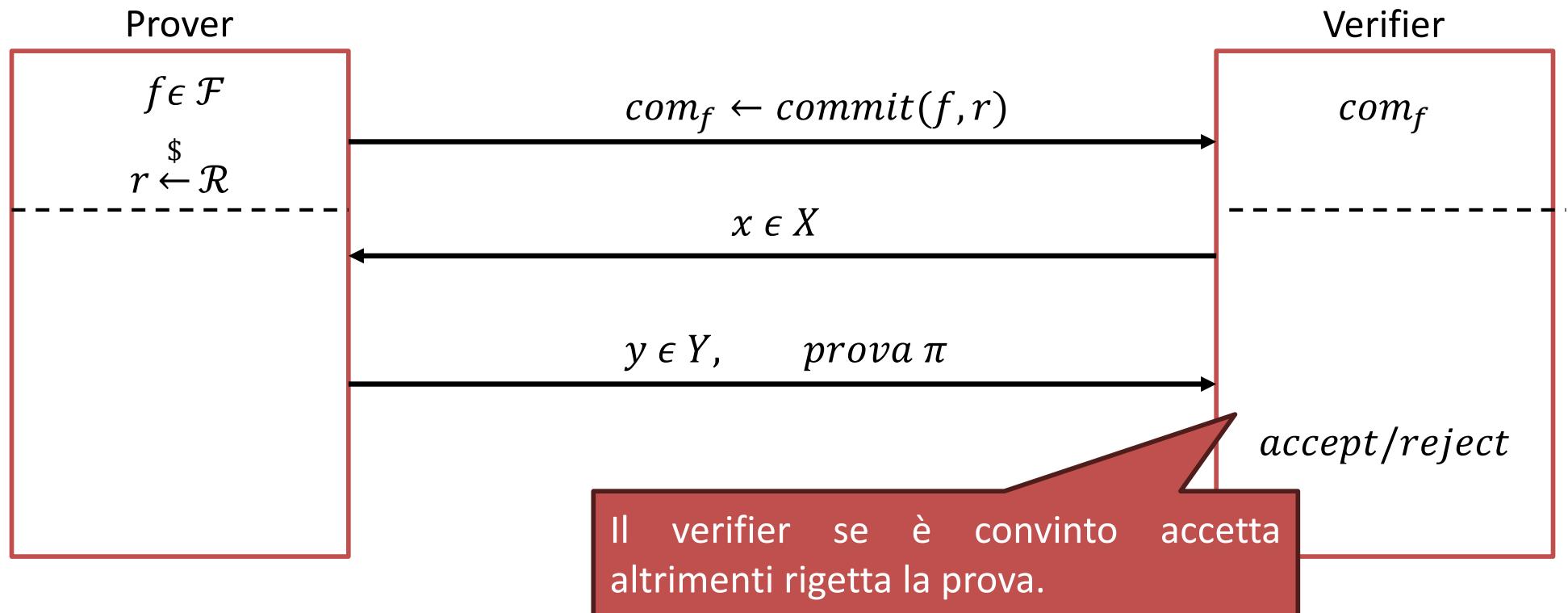
::: Schemi ZKP non interattivi (15/24)

Abbiamo bisogno di uno schema di commitment funzionale: data una famiglia di funzioni $\mathcal{F} = \{f: X \rightarrow Y\}$, lo schema lavora per funzioni in questo insieme.



::: Schemi ZKP non interattivi (15/24)

Abbiamo bisogno di uno schema di commitment funzionale: data una famiglia di funzioni $\mathcal{F} = \{f: X \rightarrow Y\}$, lo schema lavora per funzioni in questo insieme.



::: Schemi ZKP non interattivi (15/24)

Abbiamo bisogno di uno schema di commitment funzionale: data una famiglia di funzioni $\mathcal{F} = \{f: X \rightarrow Y\}$, lo schema lavora per funzioni in questo insieme.

Uno schema di commitment funzionale per \mathcal{F} si compone di

- Un algoritmo di setup $S(1^\lambda) \rightarrow gp$ che ritorna i parametri pubblici;
- Un algoritmo $commit(gp, f, r) \rightarrow com_f$ per $f \in \mathcal{F}$ e per $r \in \mathcal{R}$ che deve garantire il binding e optionalmente l'hiding (con l'hiding si può ottenere una zk-SNARK).
- Un protocollo di valutazione tra il prover e il verifier: dato una stringa com_f , per ogni statement $x \in X$ e $y \in Y$, il prover prende in ingresso i parametri pubblici gp , la funzione f e i valori x , y e r e costruisce la prova π , mentre il verifier prende gp , com_f e i valori x , y e π e decide se accettare o meno.

$eval(Prover P, Verifier V)$: $\forall com_f, x \in X \text{ e } y \in Y$:

- $P(gp, f, x, y, r) \rightarrow \pi$ (breve)
- $V(gp, com_f, x, y, \pi) \rightarrow accept/reject$

Il verifier non dispone della funzione f , ma solo del suo commitment. Questo protocollo è uno (zk-)SNARK per la relazione

$$f(x) = y, f \in \mathcal{F} \text{ e } commit(gp, f, r) = com_f$$

::: Schemi ZKP non interattivi (16/24)

Esempi di schemi di commitment funzionali:

1. Schemi di commitment polinomiale: l'insieme di tutti i polinomi a singola variabile con grado al più d $f(X)$ in $\mathbb{F}_p^{(\leq d)}[X]$.
2. Schemi di commitment multilineare: polinomi a più variabili f in $\mathbb{F}_p^{(\leq 1)}[X_1, \dots, X_k]$, e per ogni variabile il grado del polinomio è al più 1.
3. Schemi di commitment per vettori: dato un vettore $\vec{u} = (u_1, \dots, u_d) \in \mathbb{F}_p^d$, è possibile definire la funzione di accesso come $f_{\vec{u}}(i) = u_i$. A Merkle tree è un modo per implementare questo schema di commitment.
4. Schemi di commitment per il prodotto interno (generalizzazione dei primi tre): dato un vettore $\vec{u} = (u_1, \dots, u_d) \in \mathbb{F}_p^d$, si ha $f_{\vec{u}}(\vec{v}) = (\vec{u}, \vec{v})$.

Nel primo caso, il prover fa il commit per un polinomio $f(X)$ in $\mathbb{F}_p^{(\leq d)}[X]$.

- Il prover convince il verifier che per valori pubblici $u, v \in \mathbb{F}_p$ il polinomio impegnato f è tale che $f(u) = v$ e che il grado del polinomio è minore di d $\deg(f) \leq d$. Il verifier conosce solo d, u, v , e com_f .
- Siccome questa prova deve essere uno SNARK bisogna avere che la dimensione della prova e il tempo di verifica siano $O_\lambda(\log d)$.

::: Schemi ZKP non interattivi (17/24)

In letteratura esistono vari esempi pratici:

- Usando gruppi bilineari come KZG'10, con un setup fidato, oppure Dory'20, che effettua lo stesso compito senza un setup fidato ma è un po' più lento.
- Usando solo funzioni di hash: FRI, che produce delle prove relativamente lunghe.
- Usando curve ellittiche: Bulletproofs, che genera prova a lunghezza logaritmica (quindi brevi) e il tempo di verifica va come $O(d)$, quindi lineare in base al grado del polinomio (quindi lungo).
- Usando gruppi di ordine non noto: Dark'20, che non ha un setup fidato ma è relativamente lento e non ha avuto molto successo.

Quello più impiegato nella pratica è ancora KZG'10.

Lo schema di commitment banale non è uno schema polinomiale:

- $\text{commit}(f = \sum_{i=0}^d a_i X^i, r)$: ritorna $\text{com}_f \leftarrow H((a_0, \dots, a_d), r)$, ovvero l'hash dei coefficienti è sufficiente a creare il commitment per il polinomio.
- Eval: il prover invia $\pi = ((a_0, \dots, a_d), r)$ (di dimensione $d + 1$) al verifier, che accetta se $f(u) = v$ (si valuta il polinomio) e $H((a_0, \dots, a_d), r) = \text{com}_f$.

La schema non è succinto per π con la lunghezza lineare per d , come il tempo di verifica, mentre devono avere un andamento logaritmico in d .

::: Schemi ZKP non interattivi (18/24)

Alla base di tutte le costruzioni SNARK c'è un'osservazione: per una f polinomiale non-zero ($f \in \mathbb{F}_p^{(\leq d)}[X]$ e non vale sempre 0) e dato un elemento casuale nel campo $(r \xleftarrow{\$} \mathbb{F}_p)$, si ha che la probabilità che il polinomio ritorno 0 in r :

$$\Pr[f(r) = 0] \leq d/p$$



::: Schemi ZKP non interattivi (18/24)

Alla base di tutte le costruzioni SNARK c'è un'osservazione: per una f polinomiale non-zero ($f \in \mathbb{F}_p^{(\leq d)}[X]$ e non vale sempre 0) e dato un elemento casuale nel campo $(r \xleftarrow{\$} \mathbb{F}_p)$, si ha che la probabilità che il polinomio ritorno 0 in r :

$$\Pr[f(r) = 0] \leq d/p$$

Un polinomio di grado d ha al più d radici nel campo finito, e r deve essere una di queste radici. Ovvero, ci sono d radici su p possibili valori assunte da r nel campo finito, per questo la precedente probabilità assume quel valore. Ciò significa che per p scelto come un numero primo molto grande, $p \approx 2^{256}$, e il grado del polinomio grande, $d \leq 2^{40}$, allora d/p è trascurabile. Ovvero, con tali p e d la probabilità che r sia una radice del polinomio è estremamente raro.

Se si sceglie un punto casuale nel campo finito, $r \xleftarrow{\$} \mathbb{F}_p$, se $f(r) = 0$ allora è possibile dedurre che f è identicamente zero con alta probabilità. Questo fornisce un semplice test per verificare se un polinomio f è identicamente zero.

$$\text{Per } r \xleftarrow{\$} \mathbb{F}_p: \Pr[f(r) = 0] \leq d/p$$

::: Schemi ZKP non interattivi (19/24)

Lemma Schwartz-Zippel DeMillo Lipton (SZDL): La precedente diseguaglianza vale anche per polinomi multivariabile (dove d è il grado totale di f, ovvero la somma di tutti i gradi nelle variabili di f).

Si supponga di avere due polinomi f e g , entrambi con una variabile e grado al più d , e un punto random r nel campo finito \mathbb{F}_p , se i due polinomi in r assumono lo stesso valore allora sono uguali.

$$\forall f, g \in \mathbb{F}_p^{(\leq d)}[X], \forall r \xleftarrow{\$} \mathbb{F}_p : f(r) = g(r) \Rightarrow f = g$$

::: Schemi ZKP non interattivi (19/24)

Lemma Schwartz-Zippel DeMillo Lipton (SZDL): La precedente diseguaglianza vale anche per polinomi multivariabile (dove d è il grado totale di f, ovvero la somma di tutti i gradi nelle variabili di f).

Si supponga di avere due polinomi $f(r) - g(r) = 0$ e $f - g = 0$ di grado al più d, e un punto random r nel campo \mathbb{F}_p . Se i due polinomi assumono lo stesso valore allora sono uguali.

$$\forall f, g \in \mathbb{F}_p^{(\leq d)}[X], \forall r \xleftarrow{\$} \mathbb{F}_p : f(r) = g(r) \Rightarrow f = g$$

Questa relazione restituisce un semplice test di equivalenza tra polinomi.

::: Schemi ZKP non interattivi (20/24)

Prover

$$f, g \in \mathbb{F}_p^{(\leq d)}[X]$$

Verifier

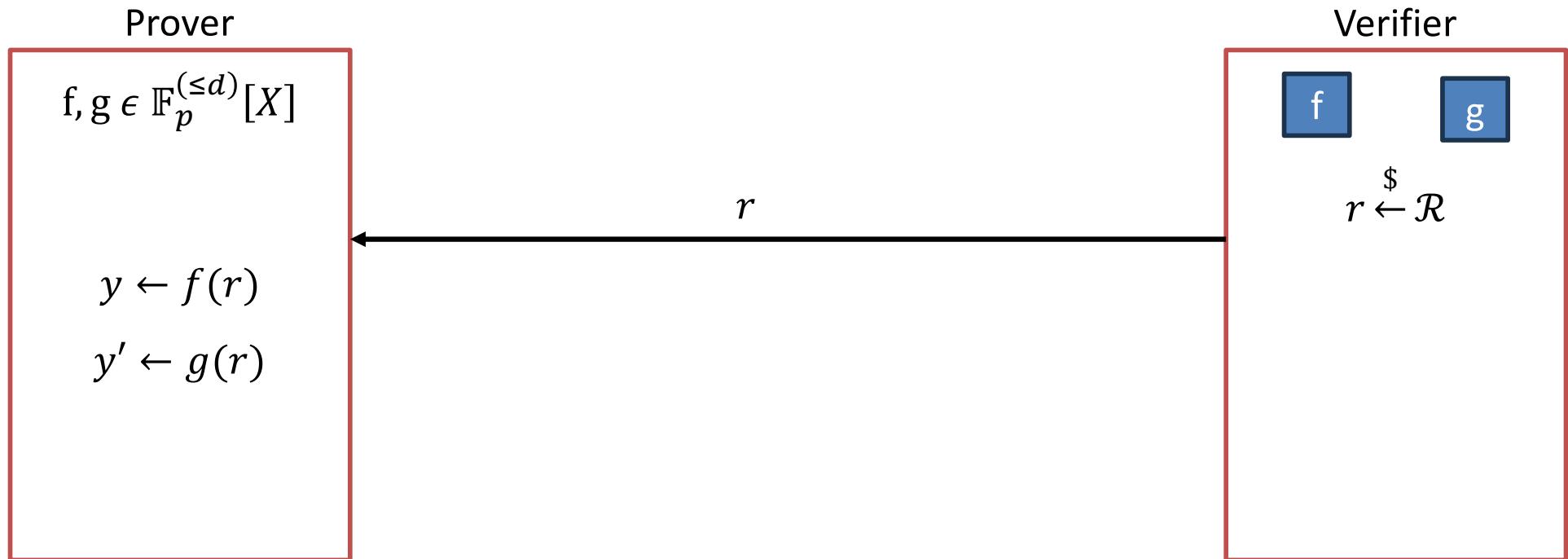
f

g

Il prover conosce due polinomi al più di grado d ad una sola variabile, mentre il verifier dispone dei commitment di questi due polinomi.

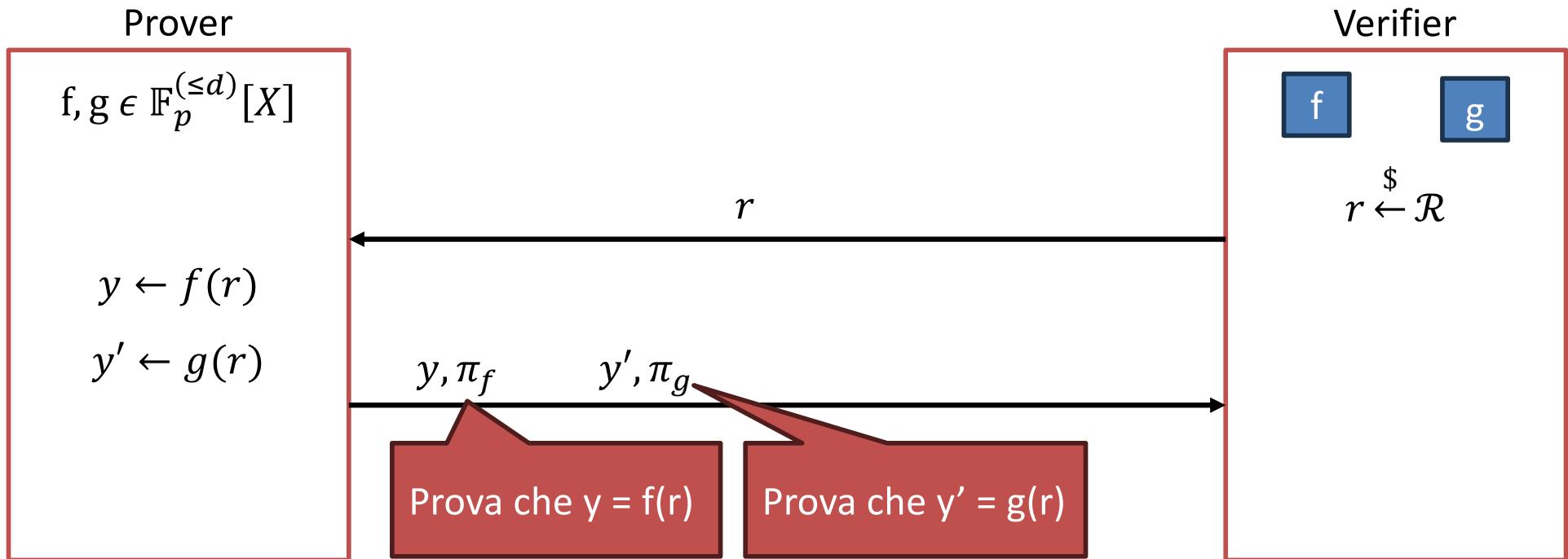
Il verifier vuole verificare che f e g siano lo stesso polinomio.

::: Schemi ZKP non interattivi (20/24)



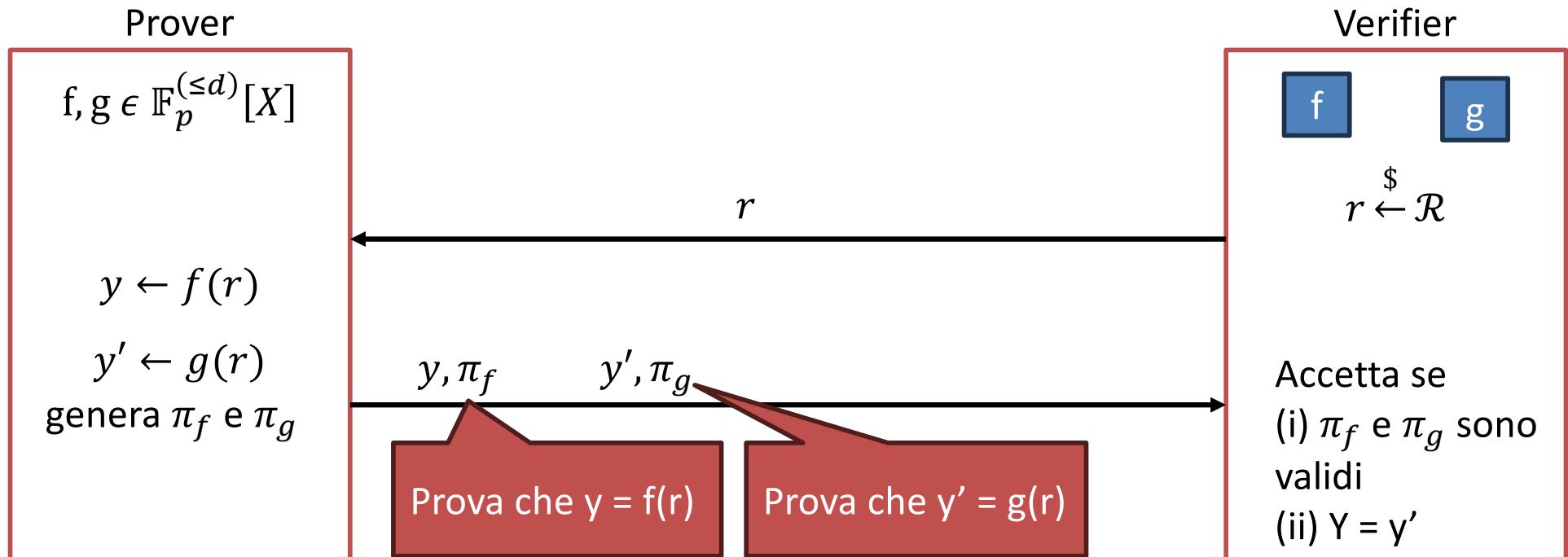
Il verifier sceglie un numero a caso nel campo finito e lo invia al prover, che verifica i due polinomi in quel punto.

::: Schemi ZKP non interattivi (20/24)



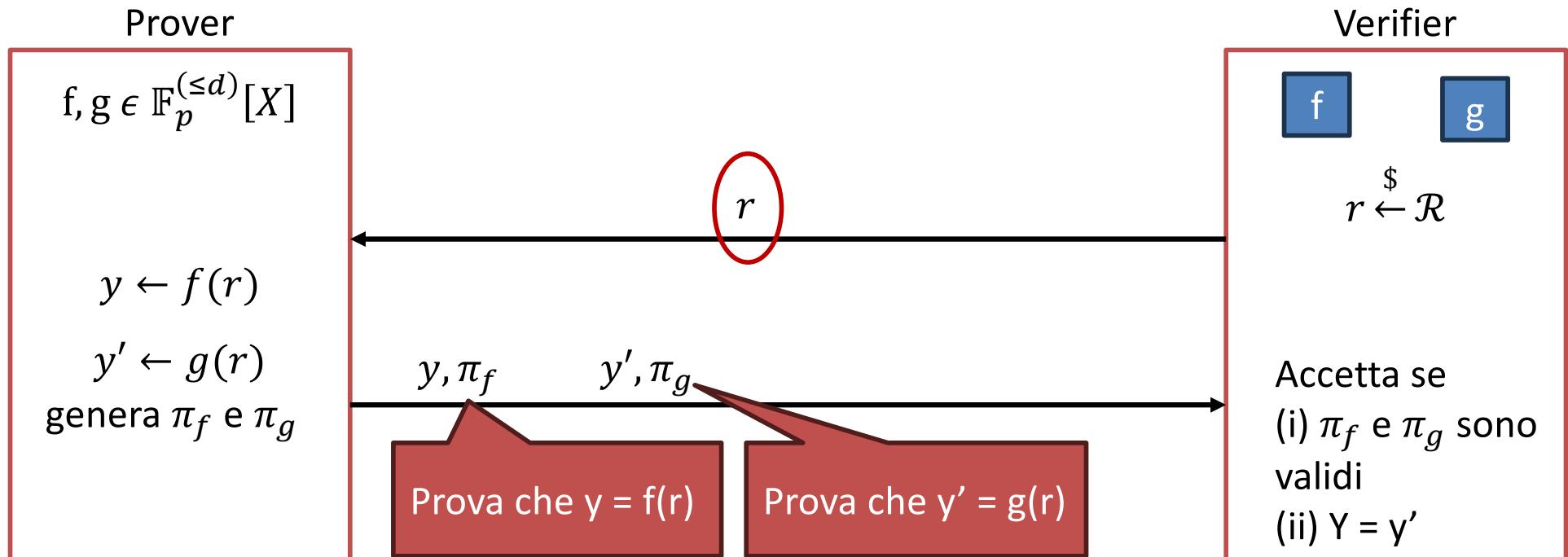
Il prover invia al verifier il risultato della valutazione del primo polinomio con la prova che y è il risultato della valutazione di f in r e il risultato della valutazione del secondo polinomio con la prova che y' è il risultato della valutazione di g in r .

::: Schemi ZKP non interattivi (20/24)



Il verifier accetta se le due prove sono valide e i due risultati identici, ovvero si convince che i due polinomi di cui ha i commitment sono identici.

::: Schemi ZKP non interattivi (20/24)

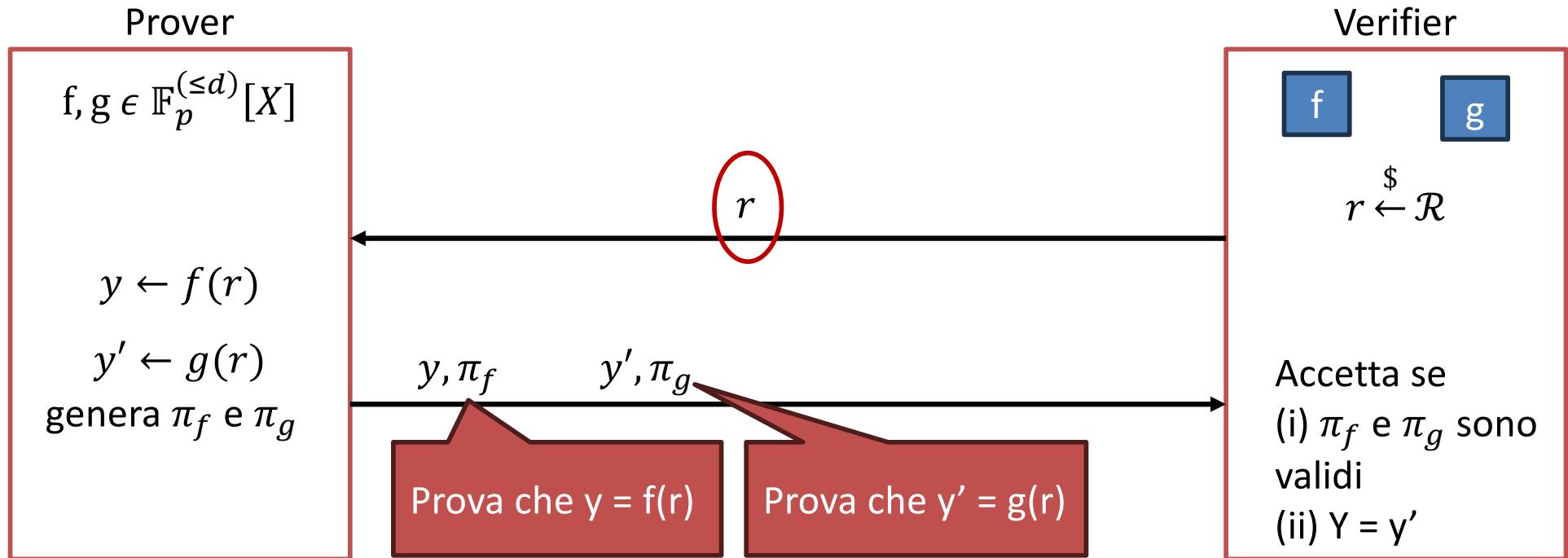


Il verifier manda solo un valore random e la risposta che ottiene è sufficiente per verificare l'affermazione.



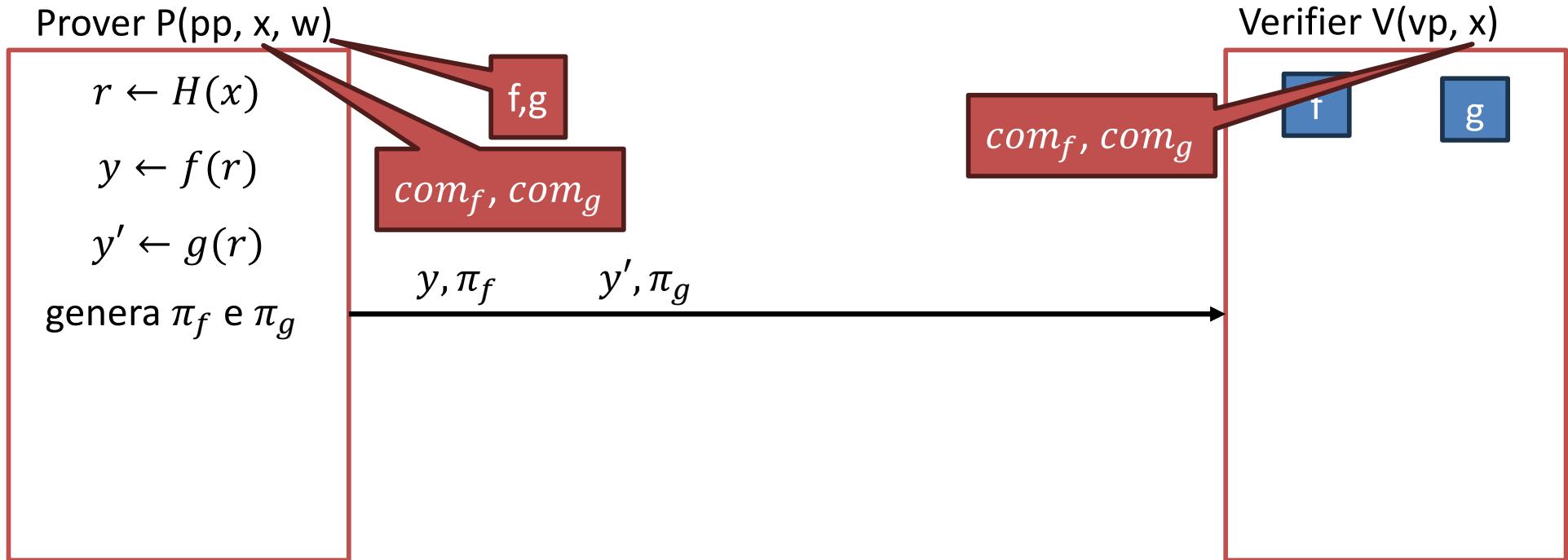
Questo protocollo è interattivo, come possiamo renderlo non interattivo?

::: Schemi ZKP non interattivi (20/24)



Possiamo renderlo non-interattivo così da avere uno SNARK per un test di uguaglianza, con la trasformazione Fiat-Shamir che dice ogni protocollo in cui il verifier manda solo dati random al prover (altrimenti detti protocolli public-coin) può essere reso non interattivo ma bisogna fare attenzione perché questa trasformazione non è sicura per ogni protocollo.

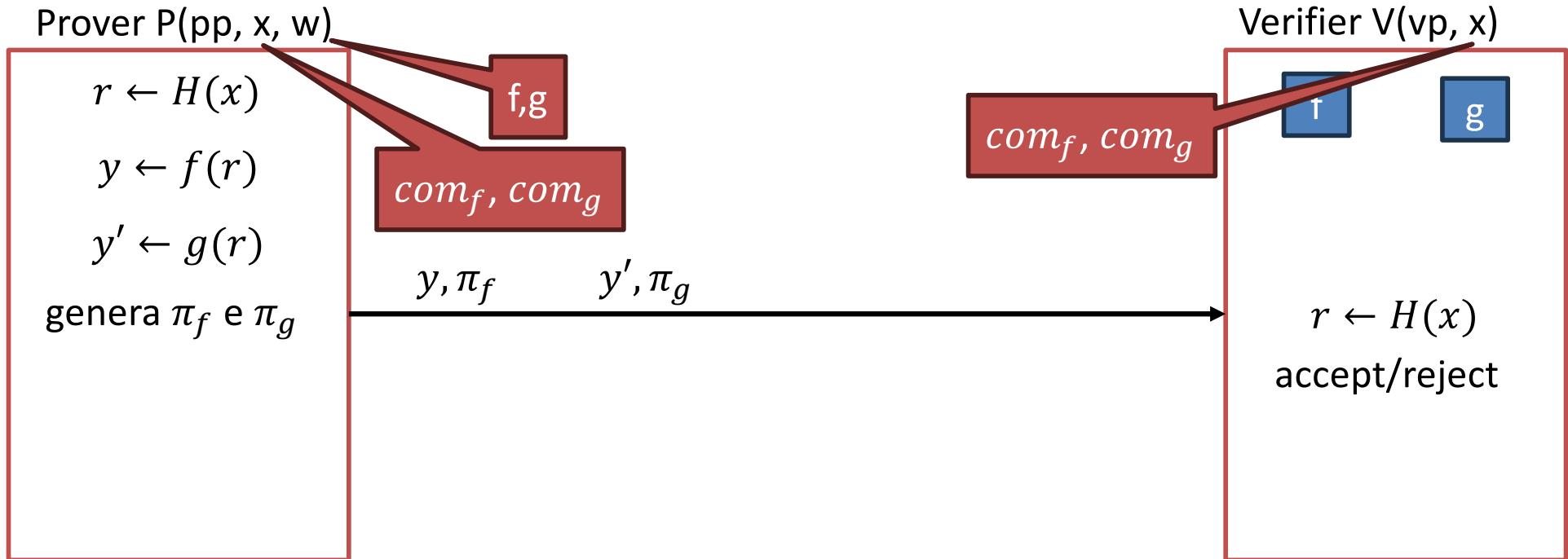
::: Schemi ZKP non interattivi (20/24)



Nello specifico, si parte da una funzione di hash crittografica, e il prover non attende il valore random dal verifier, ma lo genera da sé.

Il prover usa la funzione di hash su x ed ottenere il punto r , e continua con la sua procedura valutando i due polinomi in r e generando le due prove così da mandare il messaggio al verifier.

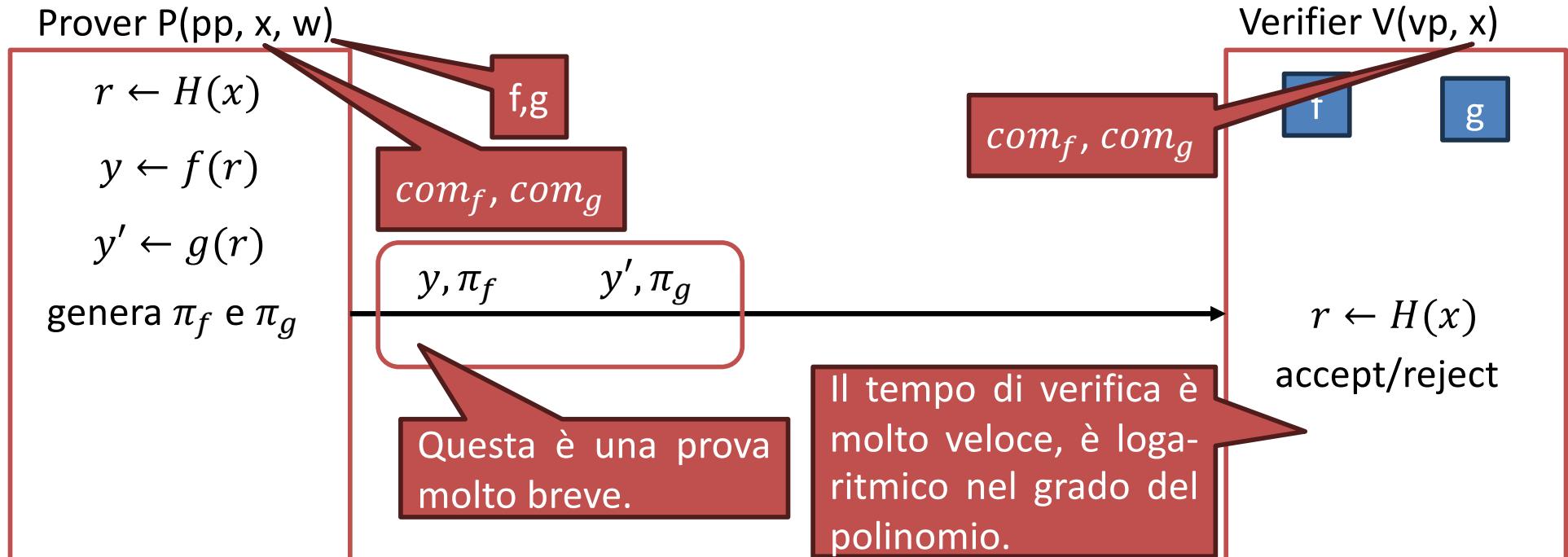
::: Schemi ZKP non interattivi (20/24)



Anche il verifier genera la challenge r con l'hash di x in suo possesso, e verifica che le risposte dal prover sono corrette.

Il protocollo è diventato non interattivo, siccome l'unico messaggio che il prover manda può essere verificato dal verifier da solo.

::: Schemi ZKP non interattivi (20/24)



Per verificare che questo protocollo è knowledge sound, basta vedere che il protocollo è uno SNARK se d/p è trascurabile (dove $f, g \in \mathbb{F}_p^{(\leq d)}[X]$) e H è modellata come un random oracle (ad es. SHA256). Non si tratta di zk-SNARK, perché il verifier apprende i valori dei polinomi f e g al punto r, ovvero apprende qualcosa che prima non sapeva.

::: Schemi ZKP non interattivi (21/24)

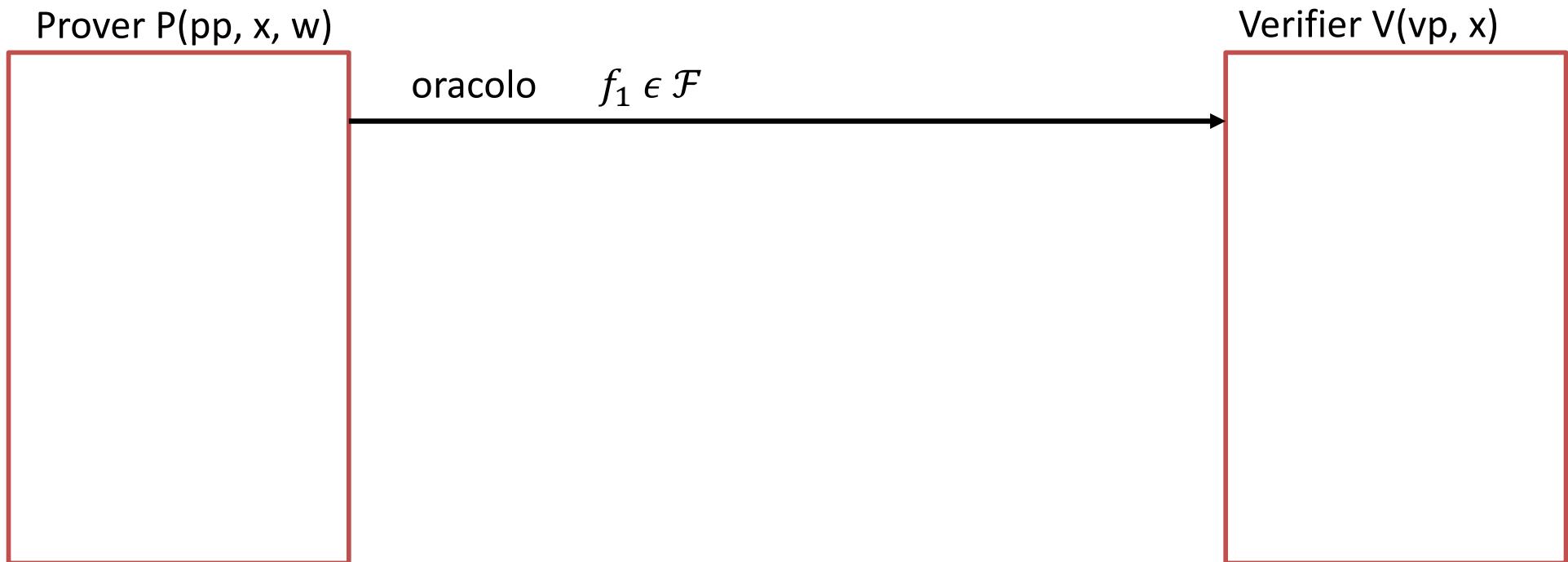
Per un generico schema SNARK per circuiti generici bisogna definire cosa si intende per una prova oracolo interattiva compatibile (IOP). Questa tecnica potenziare uno schema di commitment funzionale facendolo diventare uno schema SNARK per circuiti generali.

- Ad esempio, dato uno schema di commitment polinomiale per funzioni appartenenti all'insieme $\mathbb{F}_p^{(\leq d)}[X]$, mediante polynomial IOP si ottiene uno schema SNARK per ogni circuito aritmetico C dove $|C| < d$.

Dato un circuito $C(x,w)$ con $x \in \mathbb{F}_p^{(\leq d)}[X]$, \mathbb{F} -IOP è un sistema di prova che dimostra che il prover conosce un w che dato al circuito ritorna zero, ovvero $\exists w: C(x, w) = 0$:

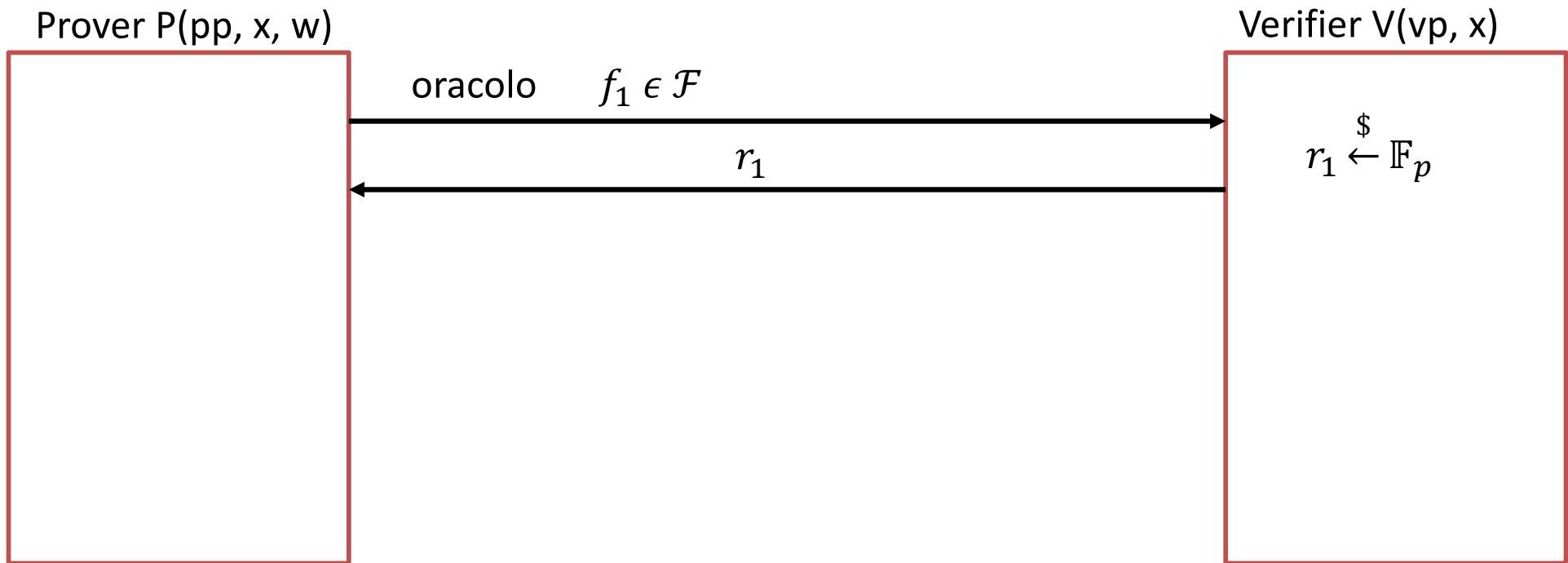
- Si esegue l'algoritmo di setup per avere i parametri pubblici: $S(C) \rightarrow (pp, vp)$, con quelli del verifier nella forma $vp = (f_0, f_{-1}, \dots, f_{-s})$ dove f_{-i} rappresenta un oracolo per funzioni nell'insieme \mathcal{F} che il verifier può interrogare, che saranno sostituite da commitments delle funzioni usando lo schema di commitment functionale che il verifier può valutare in ogni punto con l'aiuto del prover.
- Per dimostrare che il prover conosce un w tale che $C(x, w) = 0$ si ha un'interazione tra prover e verifier.

::: Schemi ZKP non interattivi (22/24)



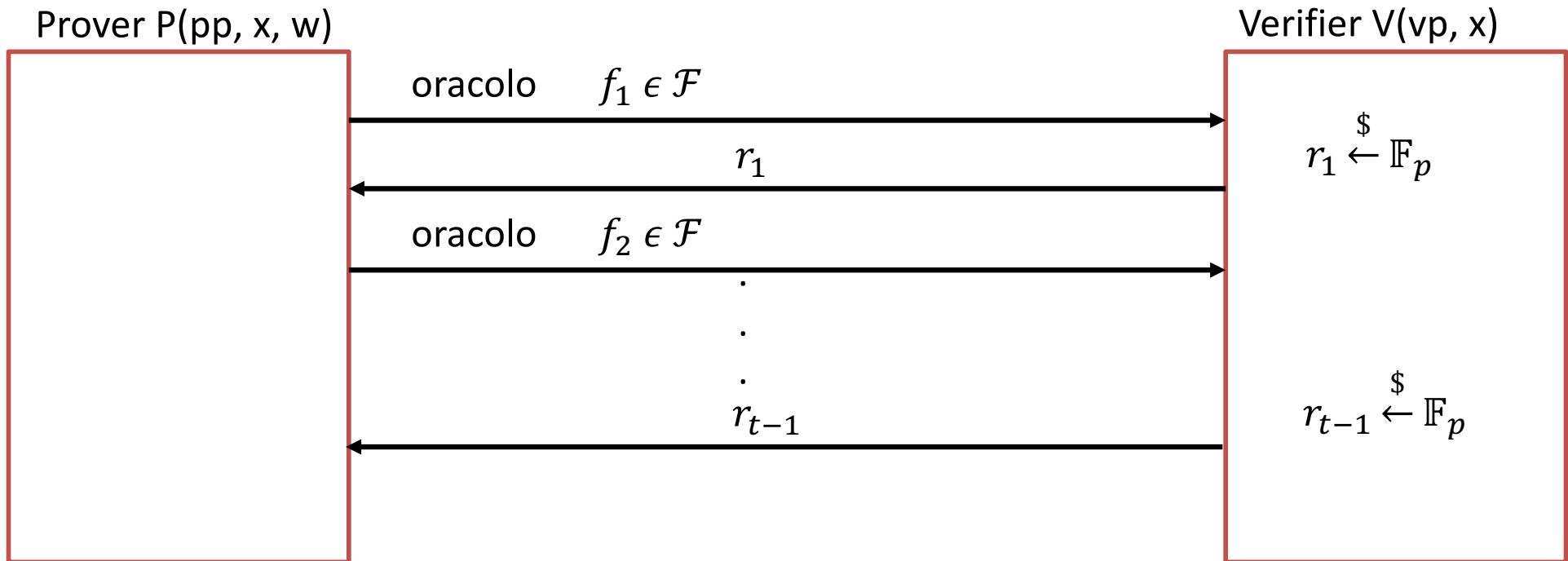
Il prover inizia l'interazione inviando la propria funzione al verifier, ovvero l'oracolo alla funzione, che successivamente il verifier può valutare in ogni punto che desidera. Quando si costruisce uno schema SNARK questo sarà sostituito da un commitment.

::: Schemi ZKP non interattivi (22/24)



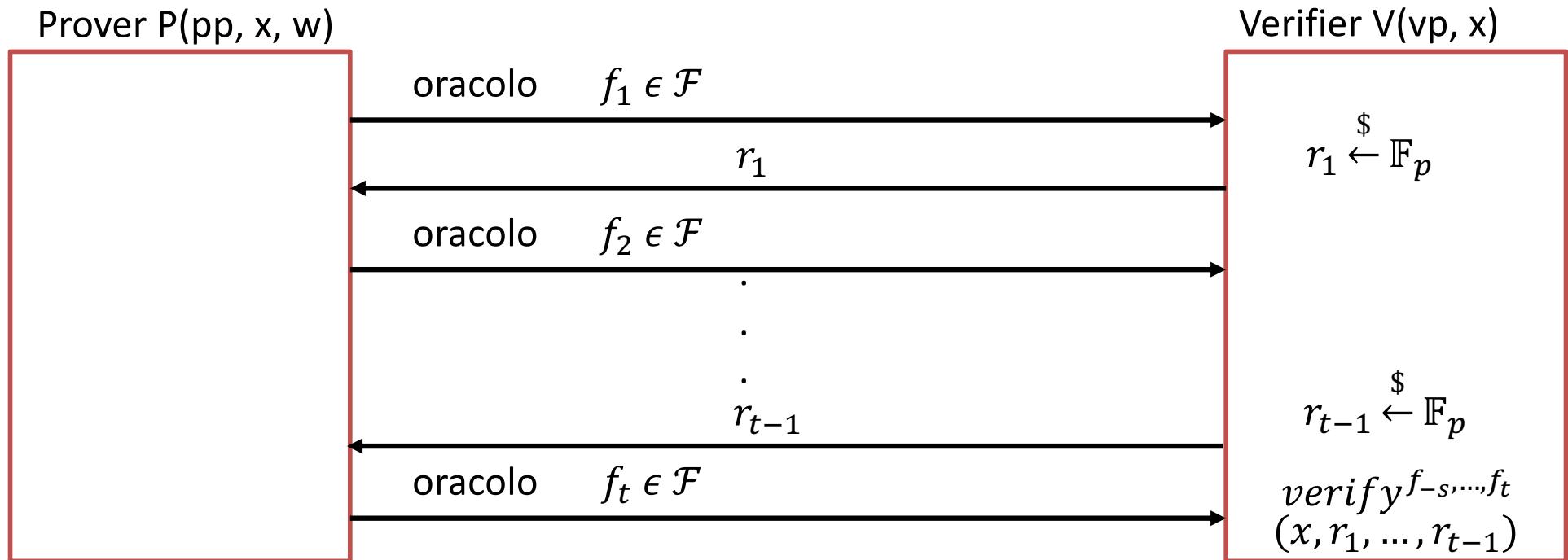
Il verifier contribuisce mandando dei dati random, ovvero scegliendo a caso un punto nel campo finito.

::: Schemi ZKP non interattivi (22/24)



Il prover manda l'oracolo per la seconda funzione, e il verifier ripete la scelta casuale per un altro punto e questo si ripete per t iterazioni fino a che il prover non manda l'ultimo oracolo.

::: Schemi ZKP non interattivi (22/24)



Il verifier deciderà di accettare o meno la prova eseguendo un'apposita procedura di verifica che prende in ingresso lo statement x e tutti i punti casuali generati e mandati al prover. La procedura considera tutti gli oracoli di funzione che sono stati inviati al verifier, ovvero sia quelli nei parametri vp che quelli avuti dall'interazione con il prover.

Nella verifica, si valutano delle funzioni casuali nei punti scelti casualmente.

::: Schemi ZKP non interattivi (22/24)

Le proprietà di un IOP sono

- Completeness: se il prover è onesto e conosce un w , allora il verifier accetterà sempre la prova, ovvero $\exists w: C(x, w) = 0 \Rightarrow \Pr[\text{accept}] = 1$.
- (Incondizionata) knowledge sound: un prover malizioso non può convincere un verifier di conoscere w tale che $C(x, w) = 0$, se non esiste un tale w o il prover non lo conosce. Questo si dimostra usando un estrattore a cui viene dato come input $(x, f_1, r_1, \dots, r_{t-1}, f_t)$, ovvero lo statement x , i punti casuali scelti dal verifier e le funzioni del prover (attenzione non gli oracoli o gli statement) e restituisce w . Siccome vengono fornite tutte le informazioni è possibile costruire estrattori che sono incondizionatamente sicuri, per questo IOT sono oggetti di teoria dell'informazione perché sono sicuri senza alcun assunzione addizionale.
- [Opzionale] zero knowledge (per un zk-SNARK): lo schema non rileva alcuna informazione rispetto al testimone w che il verifier non possiede già.

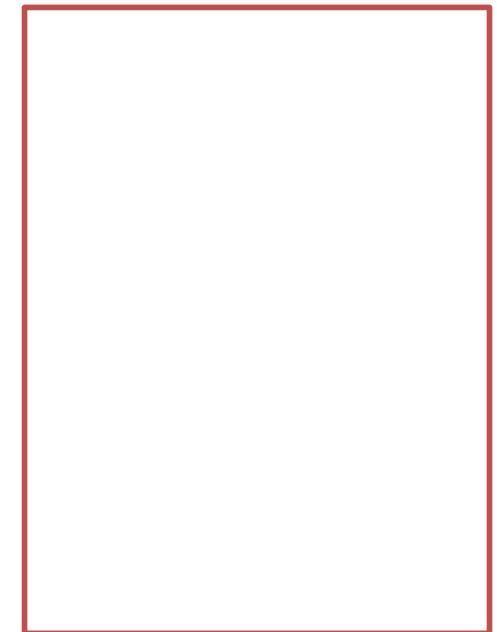
::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$



$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$



Vediamo un esempio di IOP costruito a partire da uno schema di commitment polinomiale. Il circuito che si vuole provare verifica il contenimento dove lo statement public è un determinato insieme C mentre il testimone segreto è un altro insieme W e il prover vuole dimostrare al verifier è che X è contenuto in W ed entrambi sono sottoinsiemi del campo finito \mathbb{F}_p .

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

Per questo scopo, il prover è determinare un polinomio f a singola variable (indicata con Z) costruito con tutti gli elementi dell'insieme W come sue radici.

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

Dopo, il prover costruisce un polinomio g nello stesso modo ma per l'insieme X . Questo stesso polinomio può essere costruito anche dal verifier siccome conosce l'insieme X .

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

Infine, il prover costruisce il polinomio quoziante dove si divide il polinomio f per g . Se X è contenuto in W , allora il polinomio g è un fattore del polinomio f , quindi f diviso g è un altro polinomio a singola variabile. Se invece X non è contenuto in W allora q non sarà un polinomio.

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$



Il prover invia gli oracoli per i polinomi f e q al verifier (e nel caso di uno SNARK saranno sostituiti da commitment).

::... Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

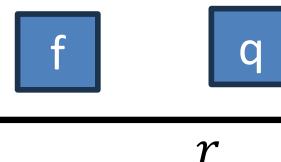
$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$r \xleftarrow{\$} \mathbb{F}_p$$



Il verifier sceglie un valore casuale r e lo invia al prover e a questo punto accetterà o rigerterà la prova eseguendo le seguenti azioni:

1. Interrogherà l'oracolo di f nel punto r : $w \leftarrow f(r)$;
2. Interrogherà l'oracolo di q nel punto r : $q' \leftarrow q(r)$;
3. Calcolerà il ritorno del polinomio g nel punto r : $x \leftarrow g(r)$;
4. Accetterà se $x \cdot q' = w$.

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$q(Z) :=$$

Nel caso di uno SNARK questo significa che il verifier chiede al prover la valutazione di f e q nel punto r e il prover risponde con le due valutazioni e con delle prove che le computazioni

Il verifier siano state svolte correttamente.

rigetterà la richiesta se seguendo le seguenti azioni:

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$r \xleftarrow{\$} \mathbb{F}_p$$

q

1. Interrogherà l'oracolo di f nel punto r : $w \leftarrow f(r)$;
2. Interrogherà l'oracolo di q nel punto r : $q' \leftarrow q(r)$;
3. Calcolerà il ritorno del polinomio g nel punto r : $x \leftarrow g(r)$;
4. Accetterà se $x \cdot q' = w$.

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

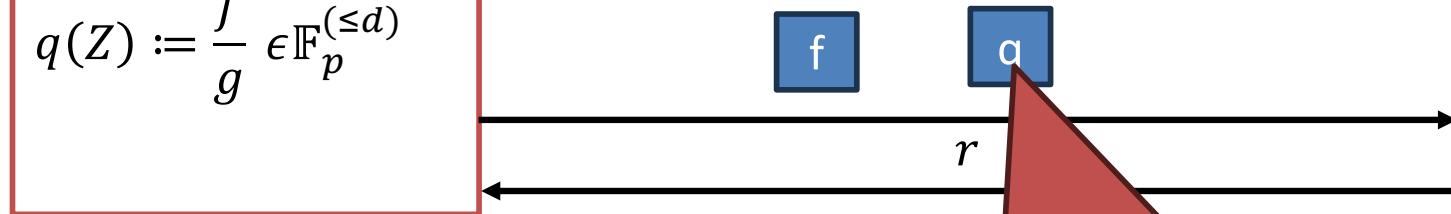
$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$r \xleftarrow{\$} \mathbb{F}_p$$



Il verifier sceglie un valore casuale r e rigetterà la prova eseguendo le seguenti

1. Interrogherà l'oracolo di f nel punto r : $f(r) \xrightarrow{\$} \mathbb{F}_p$
2. Interrogherà l'oracolo di q nel punto r : $q(r) \xrightarrow{\$} \mathbb{F}_p$
3. Calcolerà il ritorno del polinomio g nel punto r : $x \leftarrow g(r)$;
4. Accetterà se $x \cdot q'(r) = w$.

Il fatto che il prover ha inviato un oracolo per q implica che q è un polinomio, altrimenti non avrebbe potuto farlo.

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

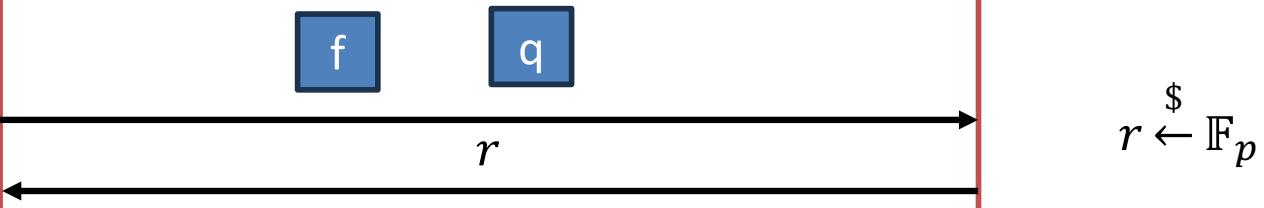
$$g(Z) := \prod_{x \in X} (Z - x)$$

$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$



Perché è un protocollo sicuro?

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

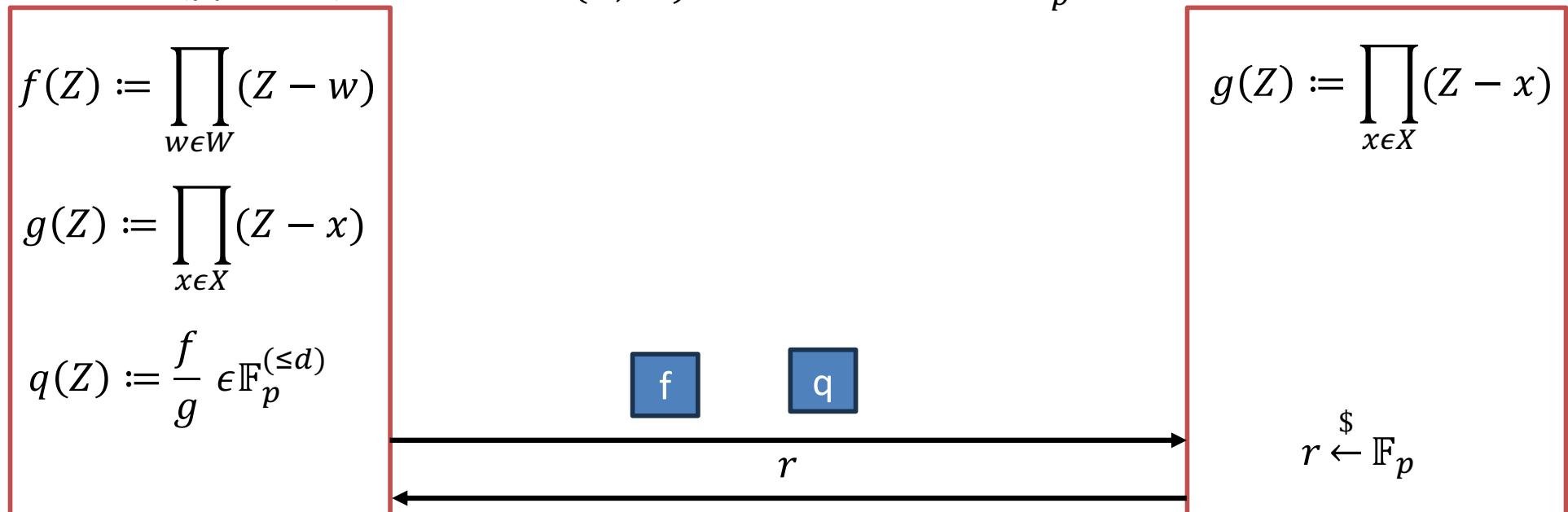
$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$r \xleftarrow{\$} \mathbb{F}_p$$



Bisogna dimostrare la knowledge soundness.

Se il verifier accetta, con alta probabilità il polinomio f è dato da $g \cdot q$. Questo è vero perché il verifier campiona i tre polinomi al punto casuale r e poi verifica che le tre valutazioni rispettino l'equazione $x \cdot q' = w$.

Questo si basa sull'osservazione che se le valutazioni di due polinomi in un punto random corrispondono allora i due polinomi sono uguali con una probabilità molto alta. Quindi g è fattore di f e ciò implica che X è sottoinsieme di W .

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

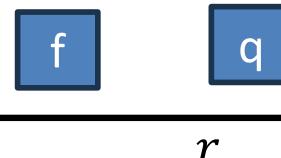
$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$r \xleftarrow{\$} \mathbb{F}_p$$



Questo non basta ma bisogna anche estrarre il testimone w dalla prova. All'estrattore si dà in input X, f, q, r ed esso eseguirà la ricerca di tutte le radici del polinomio f in Z , che corrisponderà all'insieme W , e siccome g è un fattore di f e questo significa che l'insieme X è un sottoinsieme dell'insieme estratto W .

::: Schemi ZKP non interattivi (23/24)

Prover $P(pp, X, W)$

$$f(Z) := \prod_{w \in W} (Z - w)$$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$q(Z) := \frac{f}{g} \in \mathbb{F}_p^{(\leq d)}$$

$$C(X, W) = 0 \Leftrightarrow X \subseteq W \subseteq \mathbb{F}_p$$

Verifier $V(vp, X)$

$$g(Z) := \prod_{x \in X} (Z - x)$$

$$r \stackrel{\$}{\leftarrow} \mathbb{F}_p$$



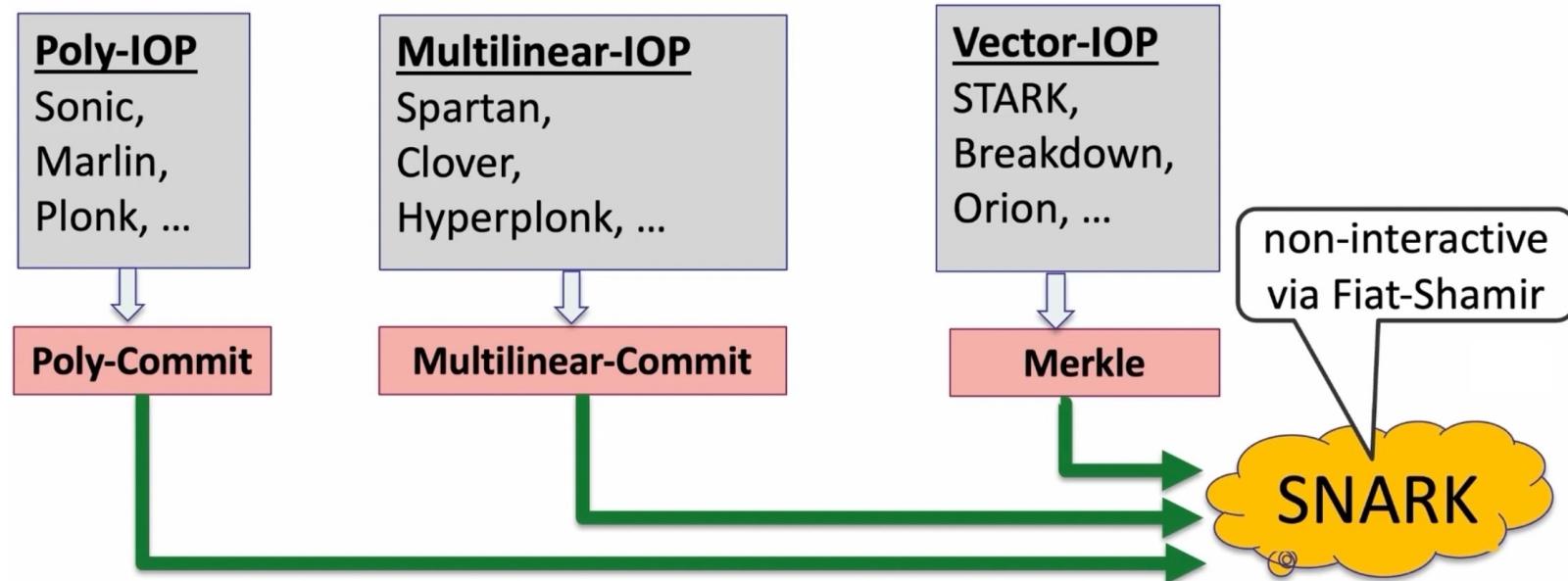
Per ottenere da questo IOP uno SNARK bisogna sostituire agli oracoli delle due funzioni i commitment e alle operazioni di interrogazione la richiesta di valutazione al prover. Questo cambiamento viene chiamato compilazione dell'IOP con lo schema di commitment polinomiale e dà come risultato uno SNARK.

::: Schemi ZKP non interattivi (24/24)

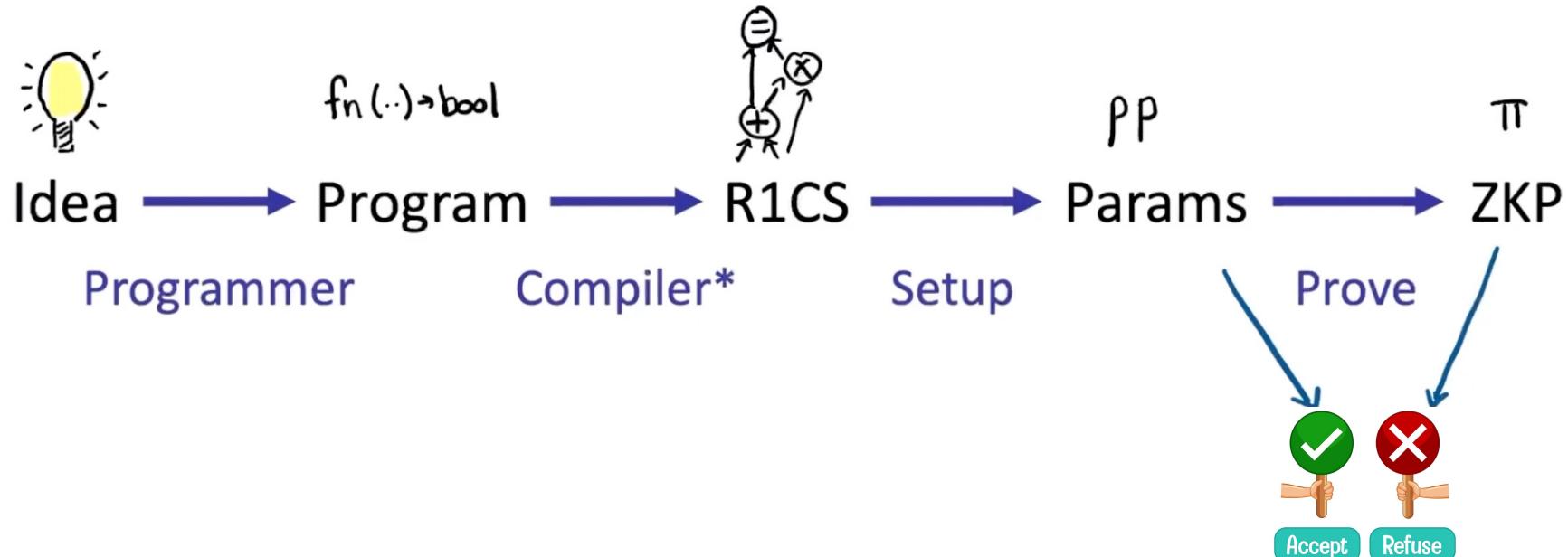
Sussistono vari esempi di IOP che possono essere impiegati per compilare vari schemi SNARK per circuiti generici:

- IOP polinomiali: Sonic, Marlin, Plonk, ...;
- IOP multilineari: Spartan, Clover, Hyperplonk, ...;
- IOP vettoriali: STARK, Breakdown, Orion, ...

Ognuno di essi possono essere compilati con lo specifico schema di commitment e la loro combinazione fondamentalmente fornisce uno SNARK. Mentre IOP è uno schema interattivo e SNARK è non-interattivo, bisogna applicare una trasformazione Fiat-Shamir come atto finale della fase di compilazione.



::: Programmazione ZKP (1/20)

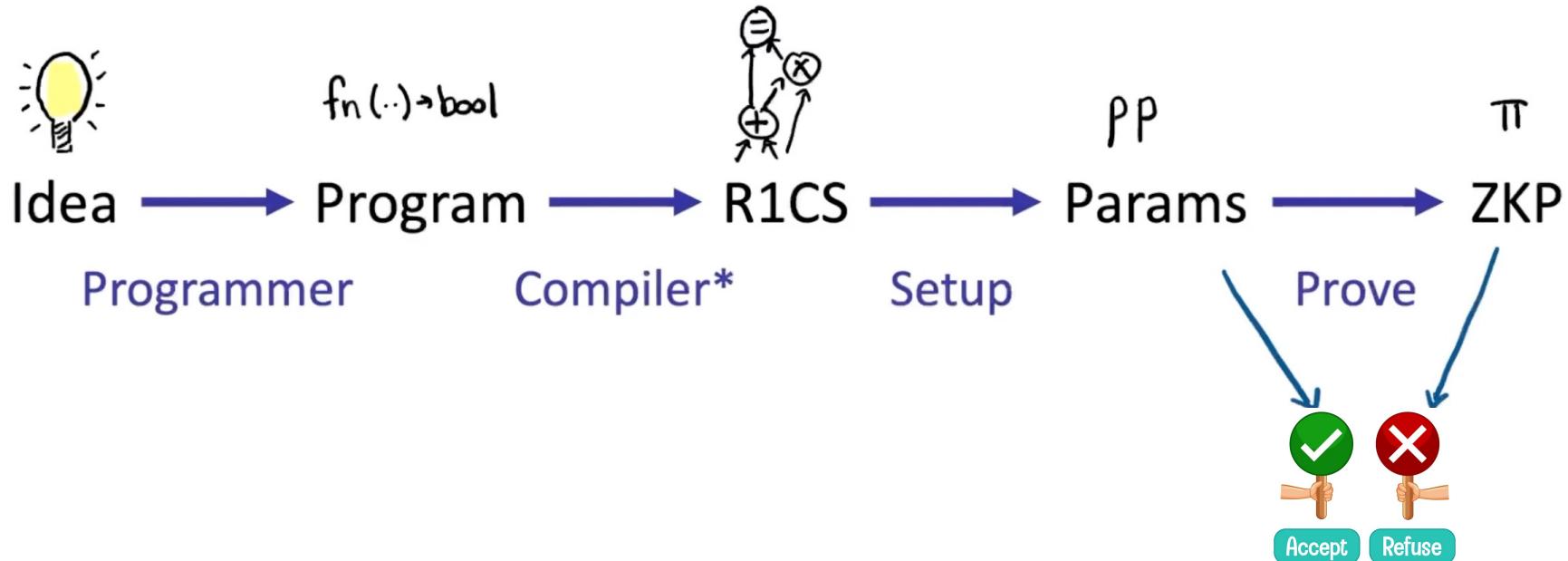


Un prover conosce un predicato ϕ , un input pubblico x noto anche al verifier e un input privato w . Lo schema ZK convince il verifier della correttezza del predicato sui due input anche se non ha a sua disposizione l'input privato.

In teoria, il predicato ϕ può essere ogni statement NP, come

- w è la fattorizzazione dell'intero x ;
- w è la chiave segreta per quella pubblica x ;
- w è la password dell'account x ;
- w è una transazione valida.

::: Programmazione ZKP (1/20)



Un prover conosce un predicato ϕ , un input pubblico x noto anche al verifier e un input privato w . Lo schema ZK convince il verifier della correttezza del predicato sui due input anche se non ha a sua disposizione l'input privato.

Nella pratica, il predicato ϕ può essere

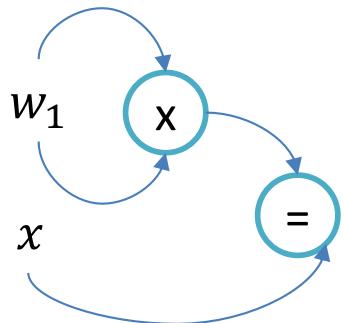
- Un circuito aritmetico sugli input x e w .

Un circuito aritmetico è simile a una rete logica con operatori booleani, ea è dato da una serie di operazioni aritmetiche (addizione +, moltiplicazione \times ed uguaglianza =) applicati agli elementi di un campo finito in p che ritornano elementi di quel campo.

::: Programmazione ZKP (2/20)

Un modo di vedere un circuito aritmetico è quello di considerarlo un sistema di equazioni definite sul campo finito in p. Esempio: $w_0 \times w_0 \times w_0 = x$ moltiplicare l'input privato per tre volte e vedere se corrisponde all'input pubblico.

È possibile esprimere ogni polinomio sul campo finito in p come un circuito aritmetico.

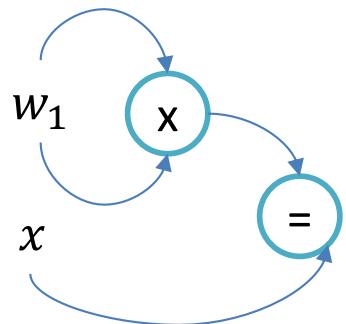


È possibile, inoltre, visualizzare un circuito aritmetico come un grafo aciclico diretto o DAG con i date come nodi intermedi e costanti/variabili come nodi foglia. L'applicazione è definita dal collegamento.

::: Programmazione ZKP (2/20)

Un modo di vedere un circuito aritmetico è quello di considerarlo un sistema di equazioni definite sul campo finito in p. Esempio: $w_0 \times w_0 \times w_0 = x$ moltiplicare l'input privato per tre volte e vedere se corrisponde all'input pubblico.

È possibile esprimere ogni polinomio sul campo finito in p come un circuito aritmetico.



È possibile, inoltre, visualizzare un circuito aritmetico come un grafo aciclico diretto o DAG con i dati come nodi intermedi e costanti/variabili come nodi foglia. L'applicazione è definita dal collegamento.

Un altro formato per esprimere un predicato che è molto in voga è Rank 1 Constraint System (R1CS), che è definito come segue:

- x è rappresentato da L elementi del campo finito, ovvero $x = (x_1, \dots, x_L)$;
- w è rappresentato da $M-L-1$ elementi, ovvero $w = (w_1, \dots, w_{M-L-1})$;
- Il predicato ϕ è definito come un insieme di n equazioni nella forma $\alpha \times \beta = \gamma$, dove α, β, γ sono combinazioni affini delle variabili, ovvero una combinazione lineare delle variabili, eventualmente con l'aggiunta di una costante.

::: Programmazione ZKP (3/20)

Esempi di predicati nel formato R1CS:

- $w_2 \times (w_3 - w_2 - 1) = x_1$

::: Programmazione ZKP (3/20)

Esempi di predicati nel formato R1CS e non:

- $\underbrace{w_2}_{\alpha} \times \underbrace{(w_3 - w_2 - 1)}_{\beta} = \underbrace{x_1}_{\gamma}$

::: Programmazione ZKP (3/20)

Esempi di predicati nel formato R1CS e non:

- $w_2 \times \underbrace{(w_3 - w_2 - 1)}_{\beta} = \underbrace{x_1}_{\gamma}$ Esempio accettabile che rispetta il formato.
- $w_2 \times w_2 = w_2$

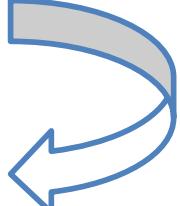
::: Programmazione ZKP (3/20)

Esempi di predicati nel formato R1CS e non:

- $w_2 \times \underbrace{(w_3 - w_2 - 1)}_{\beta} = \underbrace{x_1}_{\gamma}$ Esempio accettabile che rispetta il formato.
- $w_2 \times w_2 = w_2$ Esempio accettabile che rispetta il formato.
- $w_2 \times w_2 \times w_2 = x_1$

::: Programmazione ZKP (3/20)

Esempi di predicati nel formato R1CS e non:

- $w_2 \times \underbrace{(w_3 - w_2 - 1)}_{\beta} = \underbrace{x_1}_{\gamma}$ Esempio accettabile che rispetta il formato.
- $w_2 \times w_2 = w_2$ Esempio accettabile che rispetta il formato.
- ~~$w_2 \times w_2 \times w_2 = x_1$~~ Esempio non accettabile che rispetta il formato.
- $w_2 \times w_2 = w_4$
 $w_4 \times w_2 = x_1$ Possiamo esprimere con 2 vincoli e una variabile nuova.

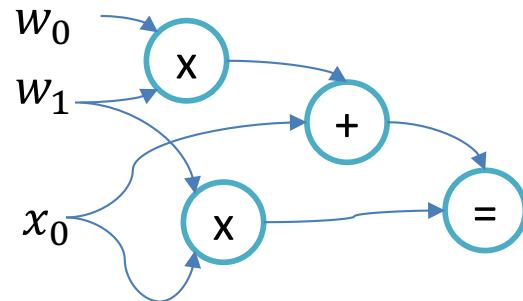
È possibile avere una rappresentazione matriciale dei predicati nel formato R1CS:

- x è rappresentato da L elementi del campo finito, ovvero $x = (x_1, \dots, x_L)$;
- w è rappresentato da $M-L-1$ elementi, ovvero $w = (w_1, \dots, w_{M-L-1})$;
- Il predicato ϕ è definito da tre matrici $A, B, C \in \mathbb{Z}_p^m$ tali che $\phi = Az \circ Bz = Cz$, dove $z = (1 \parallel |x| \parallel w) \in \mathbb{Z}_p^m$ e il simbolo ‘ \circ ’ il prodotto elemento per elemento tra le matrici e non il prodotto riga per colonna come nel resto della relazione.

È facile vedere che le righe di uguale posizione nelle tre matrici rappresentano un singolo vincolo R1CS come espresso precedentemente.

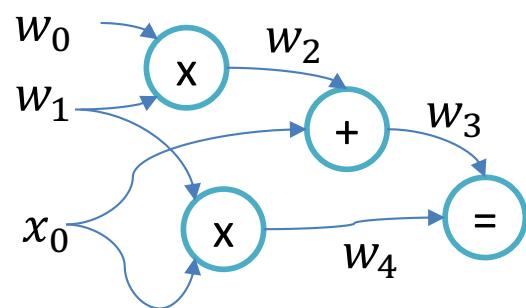
::: Programmazione ZKP (4/20)

Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



::: Programmazione ZKP (4/20)

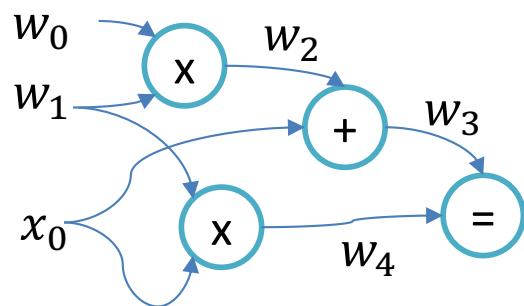
Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



- Si introduce una nuova variabile di testimone per ogni collegamento interno;

::: Programmazione ZKP (4/20)

Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



- Si introduce una nuova variabile di testimone per ogni collegamento interno;
- Si scrivono le equazioni di tipo R1CS per ogni gate nel DAG:

$$w_0 \times w_1 = w_2$$

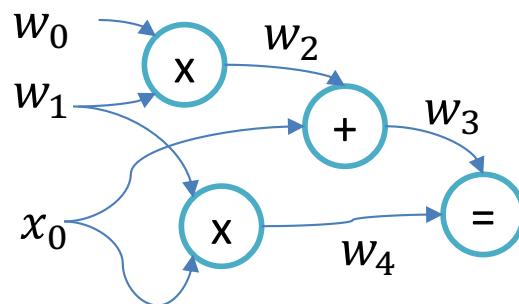
$$w_3 = w_2 + x_0 \quad \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_2 + x_0$$

$$w_1 \times x_0 = w_4$$

$$w_3 = w_4 \quad \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_4$$

::: Programmazione ZKP (4/20)

Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



- Si introduce una nuova variabile di testimone per ogni collegamento interno;
- Si scrivono le equazioni di tipo R1CS per ogni gate nel DAG:

$$w_0 \times w_1 = w_2$$

$$w_3 = w_2 + x_0 \quad \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_2 + x_0$$

$$w_1 \times x_0 = w_4$$

$$w_3 = w_4 \quad \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_4$$

Quando si vuole impiegare un sistema di prova a conoscenza zero, bisogna partire da un'idea ad altro livello e «tradurla» nella lingua parlata dal sistema, come R1CS.

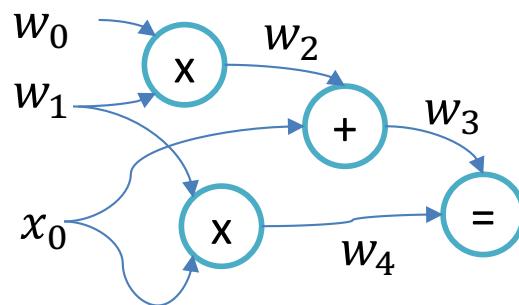
High-level
specification for ϕ



R1CS

::: Programmazione ZKP (4/20)

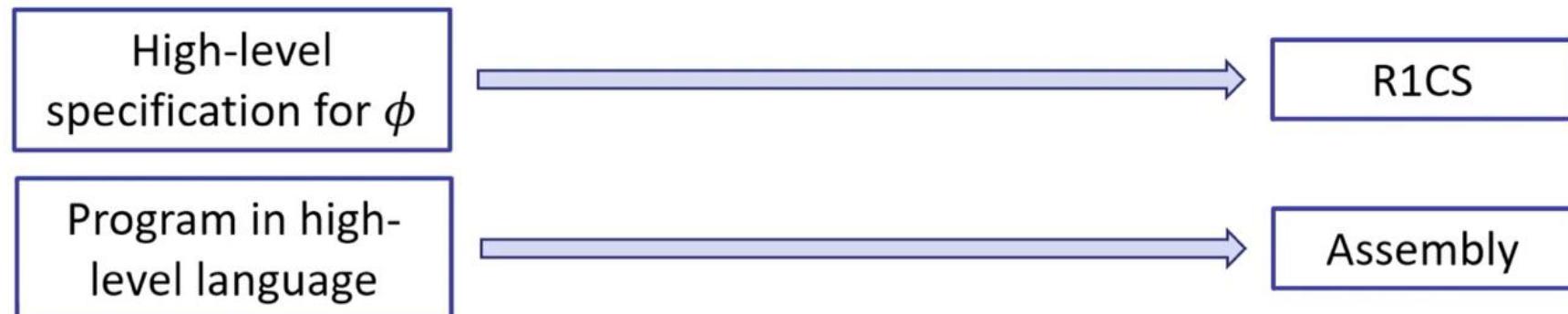
Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



- Si introduce una nuova variabile di testimone per ogni collegamento interno;
- Si scrivono le equazioni di tipo R1CS per ogni gate nel DAG:
 - $w_0 \times w_1 = w_2$
 - $w_3 = w_2 + x_0 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_2 + x_0$
 - $w_1 \times x_0 = w_4$
 - $w_3 = w_4 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_4$

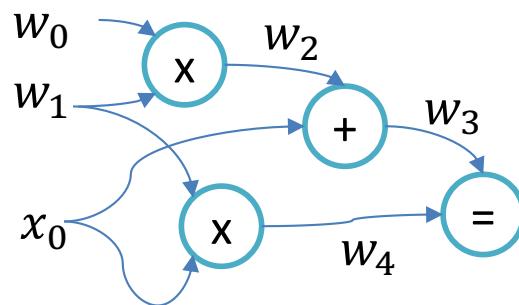
Quando si vuole impiegare un sistema di prova a conoscenza zero, bisogna partire da un'idea ad altro livello e «tradurla» nella lingua parlata dal sistema, come R1CS.

Questo processo è molto complesso.



::: Programmazione ZKP (4/20)

Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



- Si introduce una nuova variabile di testimone per ogni collegamento interno;
- Si scrivono le equazioni di tipo R1CS per ogni gate nel DAG:
 - $w_0 \times w_1 = w_2$
 - $w_3 = w_2 + x_0 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_2 + x_0$
 - $w_1 \times x_0 = w_4$
 - $w_3 = w_4 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_4$

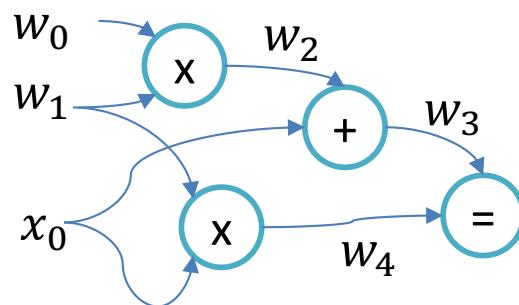
Quando si vuole impiegare un sistema di prova a conoscenza zero, bisogna partire da un'idea ad altro livello e «tradurla» nella lingua parlata dal sistema, come R1CS.

Questo processo è molto complesso, e si necessitano strumenti di supporto.



::: Programmazione ZKP (4/20)

Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



- Si introduce una nuova variabile di testimone per ogni collegamento interno;
- Si scrivono le equazioni di tipo R1CS per ogni gate nel DAG:
 - $w_0 \times w_1 = w_2$
 - $w_3 = w_2 + x_0 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_2 + x_0$
 - $w_1 \times x_0 = w_4$
 - $w_3 = w_4 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_4$

Quando si vuole impiegare un sistema di prova a conoscenza zero, bisogna partire da un'idea ad altro livello e «tradurla» nella lingua parlata dal sistema, come R1CS.

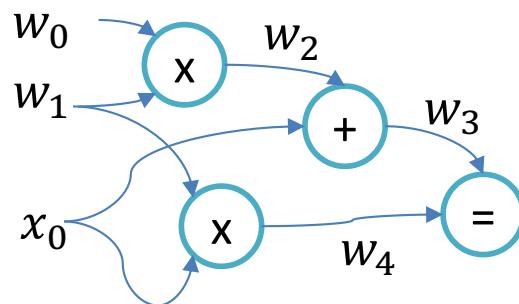
Questo processo è molto complesso, e il tipo flusso operativo è il seguente:



- Op. Booleani
- Strutture di Controllo
- Moduli/Funzioni

::: Programmazione ZKP (4/20)

Vediamo come ottenere una rappresentazione R1CS a partire dal DAG di un circuito:



- Si introduce una nuova variabile di testimone per ogni collegamento interno;
- Si scrivono le equazioni di tipo R1CS per ogni gate nel DAG:
 - $w_0 \times w_1 = w_2$
 - $w_3 = w_2 + x_0 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_2 + x_0$
 - $w_1 \times x_0 = w_4$
 - $w_3 = w_4 \Rightarrow \alpha = w_3, \beta = 1, \gamma = w_4$

Quando si vuole impiegare un sistema di prova a conoscenza zero, bisogna partire da un'idea ad altro livello e «tradurla» nella lingua parlata dal sistema, come R1CS.

Zcash è una criptovaluta con privacy e trasparenza selettiva delle transazioni.



- Alberi di Merkle
- Hash Pedersen
- Firme digitali
- Vincoli di spesa/Validità delle transazioni

::: Programmazione ZKP (5/20)

Un modo per esprimere ad alto livello il predicato ϕ è un linguaggio di specifica dell'hardware (HDL), ad esempio Circom.

- Un linguaggio di programmazione software si compone di oggetti come variabili, operazioni e funzioni e le azioni su questi oggetti sono variazioni di stato delle variabili o chiamate di funzioni.
- Un HDL nasce per la specifica di componenti hardware/circuiti, quindi si compone di collegamenti (mediante «fili»), gates o sotto-circuiti, e le azioni su questi oggetti sono la connessione di fili e la creazione di sotto-circuiti.

Circon è un HDL per R1CS e al posto dei file ha le variabili R1CS e al posto dei gate ha i vincoli R1CS. Un circuito Circom realizza due azioni: assegnare valori alle variabili e creare vincoli R1CS. Vediamo un esempio:

Un template è un (sotto)circuito.

Un «segnale» è un filo di ingresso o di uscita o una variabile R1CS.

L'operatore `<--` assegna un valore.

L'operatore `==>` crea un vincolo di rango 1:
un lato lineare e l'altro quadratico.

```
template Multiply() {
```

```
    signal input x;
```

```
    signal input y;
```

```
    signal output z;
```

```
    z <-- x * y;
```

```
    z ==> x * y;
```

```
}
```

::: Programmazione ZKP (5/20)

Un modo per esprimere ad alto livello il predicato ϕ è un linguaggio di specifica dell'hardware (HDL), ad esempio Circom.

- Un linguaggio di programmazione software si compone di oggetti come variabili, operazioni e funzioni e le azioni su questi oggetti sono variazioni di stato delle variabili o chiamate di funzioni.
- Un HDL nasce per la specifica di componenti hardware/circuiti, quindi si compone di collegamenti (mediante «fili»), gates o sotto-circuiti, e le azioni su questi oggetti sono la connessione di fili e la creazione di sotto-circuiti.

Circon è un HDL per R1CS e al posto dei file ha le variabili R1CS e al posto dei gate ha i vincoli R1CS. Un circuito Circom realizza due azioni: assegnare valori alle variabili e creare vincoli R1CS. Vediamo un esempio:

Un template è un (sotto)circuito.

Un «segnale» è un filo di ingresso o di uscita o una variabile R1CS.

Questo è un uso non valido dell'operatore.

```
template Multiply() {  
    signal input x;  
    signal input y;  
    signal output z;  
    z <-- x * y;  
    z === x * x * y;  
}
```

::: Programmazione ZKP (5/20)

Un modo per esprimere ad alto livello il predicato ϕ è un linguaggio di specifica dell'hardware (HDL), ad esempio Circom.

- Un linguaggio di programmazione software si compone di oggetti come variabili, operazioni e funzioni e le azioni su questi oggetti sono variazioni di stato delle variabili o chiamate di funzioni.
- Un HDL nasce per la specifica di componenti hardware/circuiti, quindi si compone di collegamenti (mediante «fili»), gates o sotto-circuiti, e le azioni su questi oggetti sono la connessione di fili e la creazione di sotto-circuiti.

Circon è un HDL per R1CS e al posto dei file ha le variabili R1CS e al posto dei gate ha i vincoli R1CS. Un circuito Circom realizza due azioni: assegnare valori alle variabili e creare vincoli R1CS. Vediamo un esempio:

Un template è un (sotto)circuito.

Un «segnale» è un filo di ingresso o di uscita o una variabile R1CS.

L'operatore `<==` assegna un valore e crea un vincolo di rango 1, quindi realizza entrambi i precedenti operatori.

```
template Multiply() {
```

```
    signal input x;
```

```
    signal input y;
```

```
    signal output z;
```

```
    z <== x * y;
```

```
}
```

::: Programmazione ZKP (5/20)

Un modo per esprimere ad alto livello il predicato ϕ è un linguaggio di specifica dell'hardware (HDL), ad esempio Circom.

- Un linguaggio di programmazione software si compone di oggetti come variabili, operazioni e funzioni e le azioni su questi oggetti sono variazioni di stato delle variabili o chiamate di funzioni.
- Un HDL nasce per la specifica di componenti hardware/circuiti, quindi si compone di collegamenti (mediante «fili»), gates o sotto-circuiti, e le azioni su questi oggetti sono la connessione di fili e la creazione di sotto-circuiti.

Circon è un HDL per R1CS e al posto dei file ha le variabili R1CS e al posto dei gate ha i vincoli R1CS. Un circuito Circom realizza due azioni: assegnare valori alle variabili e creare vincoli R1CS. Vediamo un esempio:

Circom sintetizza i main template, quindi bisogna dichiararne uno definendo quali sono i segnali pubblici e privati.

L'output è sempre public ma per gli input bisogna specificarli altrimenti sono considerati privati.

```
component main {public [x]} =  
    Multiply();
```



Il verifier conosce i segnali x e z di Multiply ma non y che è noto solo al prover.

::: Programmazione ZKP (6/20)

Ecco un esempio più complesso per illustrate le caratteristiche di metaprogrammazione di Circon:

Un template può avere degli argomenti, che sono fissati a tempo di compilazione.

```
template RepeatedSquaring(v) {  
    signal input x;  
    signal output y;
```

::: Programmazione ZKP (6/20)

Ecco un esempio più complesso per illustrate le caratteristiche di metaprogrammazione di Circon:

Un template può avere degli argomenti, che sono fissati a tempo di compilazione.

Gli argomenti di possono usare per dichiarare un array.

```
template RepeatedSquaring(v) {  
    signal input x;  
    signal output y;  
  
    signal xs[n+1];  
    xs[0] <== x;
```

::: Programmazione ZKP (6/20)

Ecco un esempio più complesso per illustrate le caratteristiche di metaprogrammazione di Circon:

Un template può avere degli argomenti, che sono fissati a tempo di compilazione.

Gli argomenti di possono usare per dichiarare un array, oppure si possono usare nei cicli for congiuntamente alle variabili, che non sono segnali, sono mutabili e sono valutate a tempo di compilazione (quando il compilatore processa lo script per la generazione del circuito).

```
template RepeatedSquaring(v) {  
    signal input x;  
    signal output y;  
  
    signal xs[n+1];  
    xs[0] <== x;  
  
    for(var i = 0; i < n; i++) {  
    }  
}
```

::: Programmazione ZKP (6/20)

Ecco un esempio più complesso per illustrate le caratteristiche di metaprogrammazione di Circon:

Un template può avere degli argomenti, che sono fissati a tempo di compilazione.

Gli argomenti di possono usare per dichiarare un array, oppure si possono usare nei cicli for congiuntamente alle variabili, che non sono segnali, sono mutabili e sono valutate a tempo di compilazione (quando il compilatore processa lo script per la generazione del circuito).

```
template RepeatedSquaring(v) {
    signal input x;
    signal output y;

    signal xs[n+1];
    xs[0] <== x;

    for(var i = 0; i < n; i++) {
        xs[i+1] <== xs[i] * ns[i];
    }
}
```

::: Programmazione ZKP (6/20)

Ecco un esempio più complesso per illustrate le caratteristiche di metaprogrammazione di Circon:

Un template può avere degli argomenti, che sono fissati a tempo di compilazione.

Gli argomenti di possono usare per dichiarare un array, oppure si possono usare nei cicli for congiuntamente alle variabili, che non sono segnali, sono mutabili e sono valutate a tempo di compilazione (quando il compilatore processa lo script per la generazione del circuito).

```
template RepeatedSquaring(v) {
    signal input x;
    signal output y;

    signal xs[n+1];
    xs[0] <== x;

    for(var i = 0; i < n; i++) {
        xs[i+1] <== xs[i] * ns[i];
    }
    y <== ns[n];
}
```

::: Programmazione ZKP (6/20)

Ecco un esempio più complesso per illustrate le caratteristiche di metaprogrammazione di Circon:

Un template può avere degli argomenti, che sono fissati a tempo di compilazione.

Gli argomenti di possono usare per dichiarare un array, oppure si possono usare nei cicli for congiuntamente alle variabili, che non sono segnali, sono mutabili e sono valutate a tempo di compilazione (quando il compilatore processa lo script per la generazione del circuito).

```
template RepeatedSquaring(v) {
    signal input x;
    signal output y;

    signal xs[n+1];
    xs[0] <== x;

    for(var i = 0; i < n; i++) {
        xs[i+1] <== xs[i] * ns[i];
    }
    y <== ns[n];
}

component main {public [x]} =
RepeatedSquaring(1000);
```

::: Programmazione ZKP (7/20)

Ecco un esempio che controlla che l'input non sia zero:

R1CS verificano uguaglianze non disuguaglianze, e una soluzione è di verificare se l'input ha un moltiplicativo inverso.

```
template NonZero() {  
    signal input in;
```

::: Programmazione ZKP (7/20)

Ecco un esempio che controlla che l'input non sia zero:

Questa espressione non è un vincolo di rango 1, siccome non ci sono divisioni nella sua definizione.

L'operatore '`<--`' è più generico, siccome opera un'assegnazione e non definisce un vincolo, è possibile inserire a destra ogni tipo di operazione.

```
template NonZero() {  
    signal input in;  
    signal inverse;  
  
    inverse <-- 1/ in;  
    1 === in * inverse;  
}
```

::: Programmazione ZKP (7/20)

Ecco un esempio che controlla che l'input non sia zero:

```
template NonZero() {  
    signal input in;  
    signal inverse;  
  
    inverse <- 1/in;  
    1 === in * inverse;  
}
```

```
Template Main() {  
    signal input a;  
    signal input b;  
    component nz = NonZero();  
    nz.in <== a;  
    0 === a * b;  
}
```

Questo consente di inserire un template in un altro, e di vincolare un segnale a quello di ingresso del sotto-template, riferendosi al segnale di ingresso con la notazione dot.

::: Programmazione ZKP (8/20)

Prendiamo un esempio più complesso come il gioco del Sudoku, che ha lo scopo di completare tutte le celle con valori interi, ma con i suoi vincoli:

- I valori in ogni cella devono essere compresi tra 1 e 9, inclusi;
- I valori in ogni sottomatrice 3x3 devono essere diversi
- I valori lungo ogni riga devono essere diversi
- I valori lungo ogni colonna devono essere diversi.
- La soluzione condivide gli stessi valori del puzzle iniziale, se valorizzati.

7 5	9		6	1 7 5	2 9 4	8 3 6
2 3	8		4	6 2 3	1 8 7	9 4 5
8		3	1	8 9 4	5 6 3	2 7 1
5	7 2			5 1 9	7 3 2	4 6 8
4	8 6	2		3 4 7	8 5 6	1 2 9
	9 1		3	2 8 6	9 4 1	7 5 3
9	4		7	9 3 8	4 2 5	6 1 7
6		7 5 8		4 6 1	3 7 9	5 8 2
7	1	3 9		7 5 2	6 1 8	3 9 4

Puzzle

Solution

::: Programmazione ZKP (8/20)

Prendiamo un esempio più complesso come il gioco del Sudoku, che ha lo scopo di completare tutte le celle con valori interi, ma con i suoi vincoli:

- I valori in ogni cella devono essere compresi tra 1 e 9, inclusi;
 - I valori in ogni sottomatrice 3×3 devono essere diversi
 - I valori lungo ogni riga devono essere diversi
 - I valori lungo ogni colonna devono essere diversi.
 - La soluzione condivide gli stessi valori del puzzle iniziale, se valorizzati.

7	5	9		6	1	7	5	2	9	4	8	3	6
2	3	8		4	6	2	3	1	8	7	9	4	5
8		3		1	8	9	4	5	6	3	2	7	1
5		7	2		5	1	9	7	3	2	4	6	8
	4	8	6	2	3	4	7	8	5	6	1	2	9
		9	1		2	8	6	9	4	1	7	5	3
9		4		7	9	3	8	4	2	5	6	1	7
	6		7	5 8	4	6	1	3	7	9	5	8	2
7		1	3	9	7	5	2	6	1	8	3	9	4

Puzzle

Solution

→ Predicato pubblico ϕ

Statement x pubblico

Testimone w privato

Si vuole un sistema di prova per convincere della conoscenza della soluzione w , dato come pubblico il predicato e il puzzle iniziale.

::: Programmazione ZKP (9/20)

```
pragma circom 2.0.0;

template NonEqual(){
    signal input in0;
    signal input in1;
    // check that (in0 - in1) is non-zero.
    signal inverse;
    inverse <-- 1 / (in0 - in1);
    inverse*(in0 - in1) === 1;
}
```

```
template Distinct(n) {
    signal input in[n];
    component nonEqual[n][n];
    for (var i = 0; i < n; i++) {
        for (var j = 0; j < i; j++) {
            nonEqual[i][j] = NonEqual();
            nonEqual[i][j].in0 <= in[i];
            nonEqual[i][j].in1 <= in[j];
        }
    }
}
```

```
template Bits4(){
    signal input in;
    signal bits[4];
    var bitsum = 0;
    for (var i = 0; i < 4; i++) {
        bits[i] <-- (in >> i) & 1;
        bits[i] * (bits[i] - 1) === 0;
        bitsum = bitsum + 2 ** i * bits[i];
    }
    bitsum === in;
```

Template in Circom per testare la non uguaglianza di due valori dati come input.

Template in Circom per testare che tutti gli elementi in un array dato come input sono distinti.

Template in Circom per testare che un valore sia nell'intervallo da 0 a 15, inclusi.

::: Programmazione ZKP (10/20)

```
template OneToNine() {
    signal input in;
    component lowerBound = Bits4();
    component upperBound = Bits4();
    lowerBound.in <== in - 1;
    upperBound.in <== in + 6;
}
```

Template in Circom per testare che un valore sia compreso tra 1 e 9.

```
template Sudoku(n) {
    // solution is a 2D array: indices are (row_i, col_i)
    signal input solution[n][n];
    // puzzle is the same, but a zero indicates a blank
    signal input puzzle[n][n];

    // ensure that each solution # is in-range.
    component inRange[n][n];
    for (var i = 0; i < n; i++) {
        for (var j = 0; j < n; j++) {
            inRange[i][j] = OneToNine();
            inRange[i][j].in <== solution[i][j];
        }
    }

    // ensure that puzzle and solution agree.
    for (var i = 0; i < n; i++) {
        for (var j = 0; j < n; j++) {
            // puzzle_cell * (puzzle_cell - solution_cell) === 0
            puzzle[i][j] * (puzzle[i][j] - solution[i][j]) === 0
        }
    }

    // ensure uniqueness in ROWs.
    component distinct[n];
    for (var i = 0; i < n; i++) {
        distinct[i] = Distinct(n);
        for (var j = 0; j < n; j++) {
            distinct[i].in[j] <== solution[i][j];
        }
    }

    component main {public[puzzle]} = Sudoku(9);
```

Template in Circom per realizzare il predicato del gioco del Sudoku (non realizzando la verifica per colonne e sottomatrice 3x3, che sono riconducibili al controllo fatto per riga).

::: Programmazione ZKP (11/20)

1. Si progetta il circuito aritmetico e lo si scrive usando Circom, scrivendo del codice oppure impiegando dei template safe da apposite librerie;
2. Il circuito viene compilato per avere una rappresentazione di basso livello in R1CS:

```
$ circom circuit.circom --r1cs --wasm --sym
```

3. Si usa snarkjs per computare il testimone:

```
$ snarkjs calculateWitness --wasm circuit.wasm --input input.json --witness witness.json
```

4. Si genera un setup fidato e si ottiene la prova zk-SNARK:

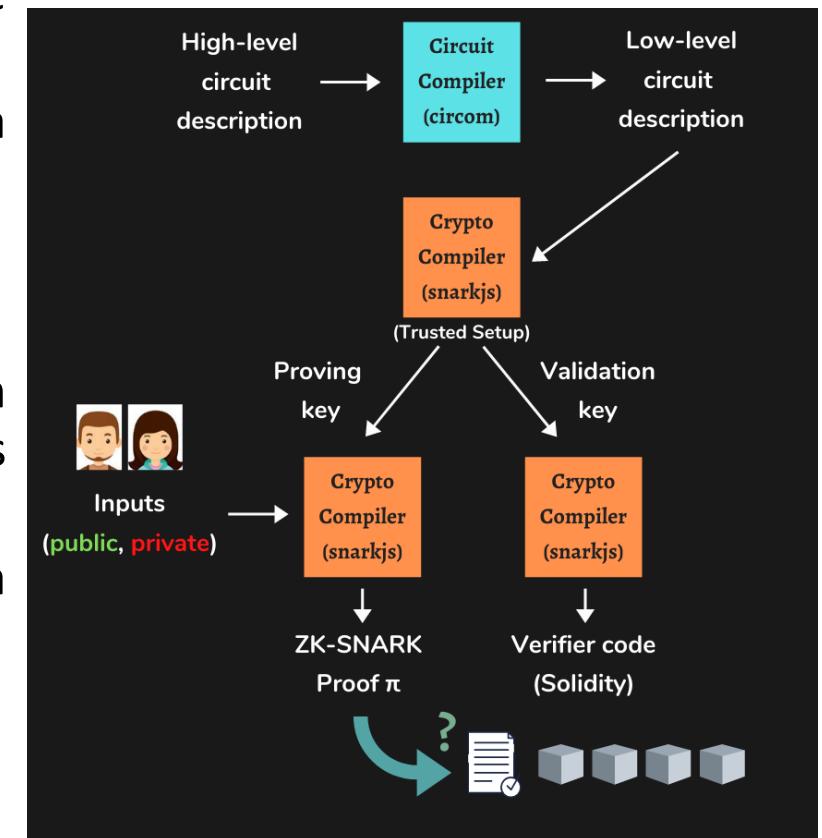
```
$ snarkjs setup
```

```
$snarkjs proof
```

5. Si valida la prova o si ha uno smart-contrat per validarla:

```
$ snarkjs validate
```

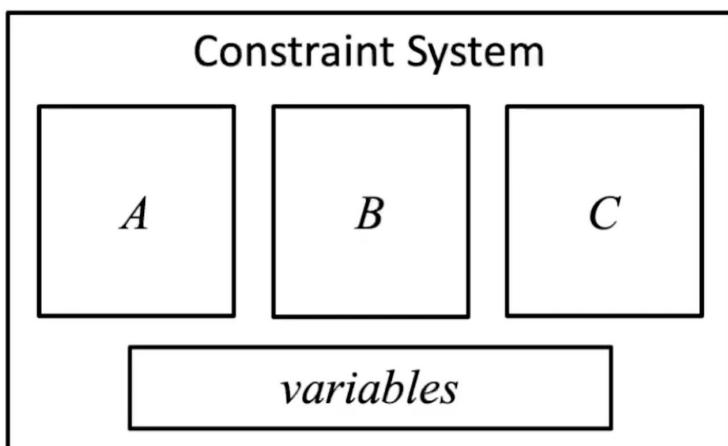
```
$ snarkjs generateVerifier
```



::: Programmazione ZKP (12/20)

Il secondo modo per programmare sistemi di prova, e in particolare R1CS, è detto approccio di libreria. Un HDL per specificare entità in R1CS, come Circom, dà allo sviluppatore il diretto controllo sui vincoli e la loro specifica, ma si tratta di un linguaggio specifico e questo ha luci ed ombre in base a quanto l'HDL è stato ben progettato. Un modo alternativo consiste di progettare una libreria, in un determinato linguaggio (tipicamente un linguaggio di alto livello), con un solo tipo chiave, ovvero il sistema di vincoli.

- Questo oggetto è responsabile di mantenere lo stato del sistema di vincoli R1CS costruito al di sopra, e le variabili e i loro valori associati.
- Il sistema internamente mantiene la rappresentazione delle tre matrici del sistema di vincolo, ovvero A, B e C, come anche i valori associati alle variabili.

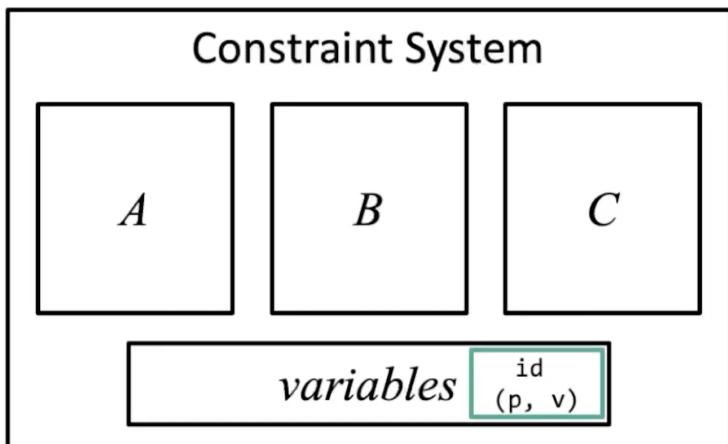


- Le operazioni che si possono svolgere sul sistema sono (i) creare una nuova variabile, come aggiunta alla lista delle variabili, (ii) creare combinazioni lineari delle variabili, (iii) aggiungere un vincolo, come nuove righe nelle tre matrici.

::: Programmazione ZKP (12/20)

Il secondo modo per programmare sistemi di prova, e in particolare R1CS, è detto approccio di libreria. Un HDL per specificare entità in R1CS, come Circom, dà allo sviluppatore il diretto controllo sui vincoli e la loro specifica, ma si tratta di un linguaggio specifico e questo ha luci ed ombre in base a quanto l'HDL è stato ben progettato. Un modo alternativo consiste di progettare una libreria, in un determinato linguaggio (tipicamente un linguaggio di alto livello), con un solo tipo chiave, ovvero il sistema di vincoli.

- Questo oggetto è responsabile di mantenere lo stato del sistema di vincoli R1CS costruito al di sopra, e le variabili e i loro valori associati.
- Il sistema internamente mantiene la rappresentazione delle tre matrici del sistema di vincolo, ovvero A, B e C, come anche i valori associati alle variabili.

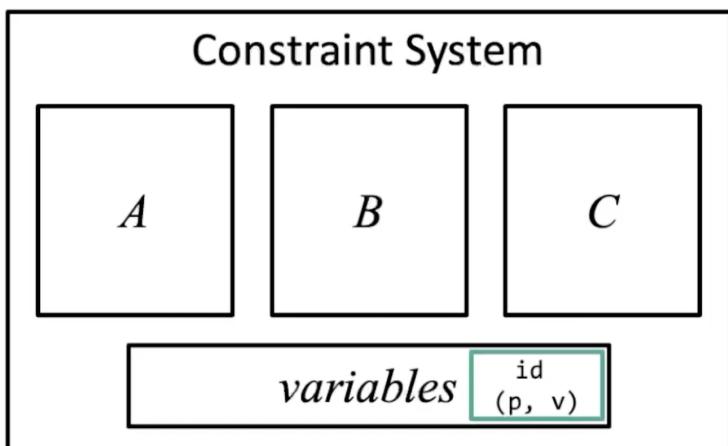


- Vediamo in Rust come (i) creare una nuova variabile, come aggiunta alla lista delle variabili:
`cs.add_var(p, v) -> id`
 - cs è l'oggetto del sistema di vincoli;
 - p è la visibilità della variabile;
 - v è il valore assegnato;
 - id è l'handle della variabile.

::: Programmazione ZKP (12/20)

Il secondo modo per programmare sistemi di prova, e in particolare R1CS, è detto approccio di libreria. Un HDL per specificare entità in R1CS, come Circom, dà allo sviluppatore il diretto controllo sui vincoli e la loro specifica, ma si tratta di un linguaggio specifico e questo ha luci ed ombre in base a quanto l'HDL è stato ben progettato. Un modo alternativo consiste di progettare una libreria, in un determinato linguaggio (tipicamente un linguaggio di alto livello), con un solo tipo chiave, ovvero il sistema di vincoli.

- Questo oggetto è responsabile di mantenere lo stato del sistema di vincoli R1CS costruito al di sopra, e le variabili e i loro valori associati.
- Il sistema internamente mantiene la rappresentazione delle tre matrici del sistema di vincolo, ovvero A, B e C, come anche i valori associati alle variabili.

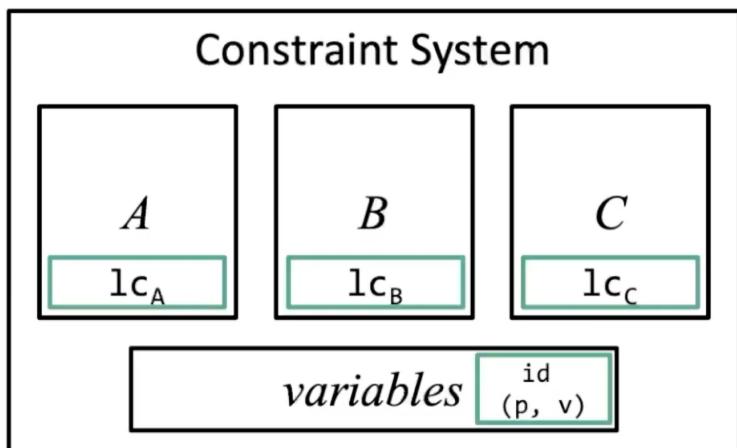


- Vediamo in Rust come (ii) creare combinazioni lineari delle variabili:
`cs.zero() -> lc`
`lc.add (c, id) -> lc'`
 - id è l'handle della variabile;
 - c è il coefficiente
 - $lc' := lc + c * id$.

::: Programmazione ZKP (12/20)

Il secondo modo per programmare sistemi di prova, e in particolare R1CS, è detto approccio di libreria. Un HDL per specificare entità in R1CS, come Circom, dà allo sviluppatore il diretto controllo sui vincoli e la loro specifica, ma si tratta di un linguaggio specifico e questo ha luci ed ombre in base a quanto l'HDL è stato ben progettato. Un modo alternativo consiste di progettare una libreria, in un determinato linguaggio (tipicamente un linguaggio di alto livello), con un solo tipo chiave, ovvero il sistema di vincoli.

- Questo oggetto è responsabile di mantenere lo stato del sistema di vincoli R1CS costruito al di sopra, e le variabili e i loro valori associati.
- Il sistema internamente mantiene la rappresentazione delle tre matrici del sistema di vincolo, ovvero A, B e C, come anche i valori associati alle variabili.



- Vediamo in Rust come (iii) aggiungere un vincolo, come nuove righe nelle tre matrici:
`cs.constrain(lcA, lcB, lcC)`
 - Aggiunge un vincolo espresso come $lc_A \times lc_B = lc_C$.

::: Programmazione ZKP (12/20)

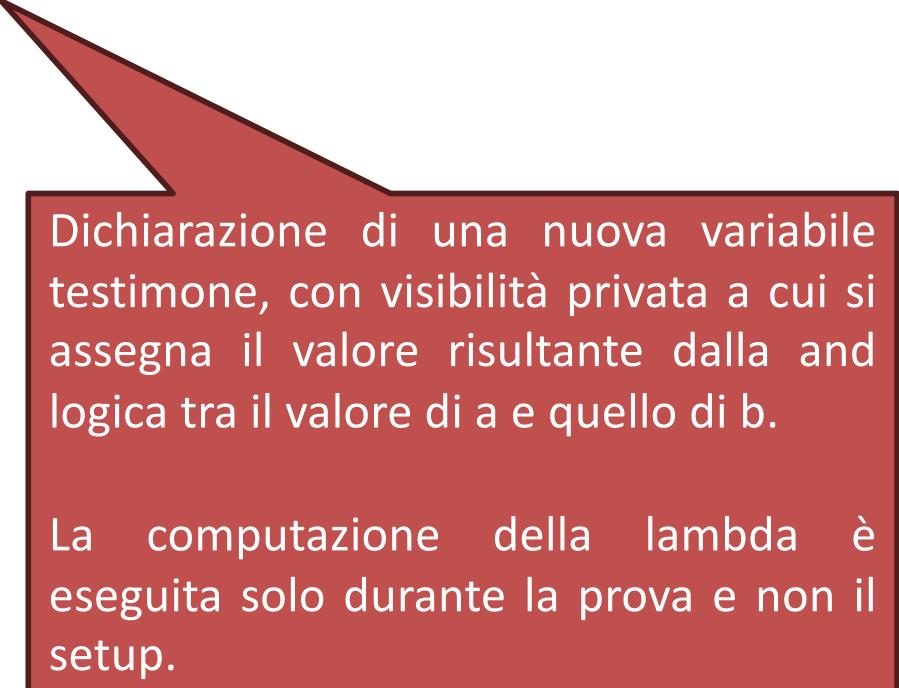
Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Vat ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}
```

::: Programmazione ZKP (12/20)

Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Vat ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}
```



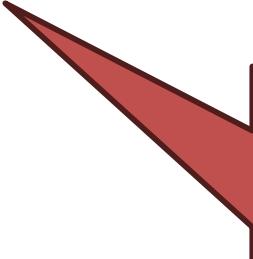
Dichiarazione di una nuova variabile testimone, con visibilità privata a cui si assegna il valore risultante dalla and logica tra il valore di a e quello di b.

La computazione della lambda è eseguita solo durante la prova e non il setup.

::: Programmazione ZKP (12/20)

Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Vat ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}
```



Impostazione che il risultato rispetta il vincolo che sia la and di a e b: vengono create tre combinazioni lineari, ognuna con una variabile, e usate per creare un vincolo R1CS ovvero la moltiplicazione di a e b deve essere pari a result.

::: Programmazione ZKP (12/20)

Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Vat ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}
```

Questo stile di programmazione è abbastanza tedioso e complesso, e non si presta per la definizione di complessi sistemi di prova come la verifica di una firma. Si preferisce usare la potenza del linguaggio usato, Rust, per astrarre i dettagli di basso livello.

::: Programmazione ZKP (12/20)

Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Var ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}  
struct Boolean { var: Var };  
impl BitAnd for Boolean {  
    fn and(self: Boolean, Other: Boolean) -> Boolean {  
        //come prima  
        Boolean [ var: result ]  
    } }  

```

::: Programmazione ZKP (12/20)

Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Vat ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}  
  
struct Boolean { var: Var };  
impl BitAnd for Boolean {  
    fn and(self: Boolean, Other: Boolean) -> Boolean {  
        //come prima  
        Boolean [ var: result ]  
    } }  
}
```

Racchiudere una variabile in un tipo dedicato.

::: Programmazione ZKP (12/20)

Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Var ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}  
struct Boolean { var: Var };  
impl BitAnd for Boolean {  
    fn and(self: Boolean, Other: Boolean) -> Boolean {  
        //come prima  
        Boolean [ var: result ]  
    } }  


Implementare l'interfaccia per l'overloading degli operatori.


```

::: Programmazione ZKP (12/20)

Vediamo un esempio di realizzare una AND booleana:

```
fn and(cs: ConstraintSystem, a: Var, b: Var ) -> Var {  
    let result = cs.new_witness_var(|| a.value() & b.value());  
    self.cs.enforce_constraint(  
        lc!() + a,  
        lc!() + b,  
        lc!() + result,  
    );  
    result  
}  
  
struct Boolean { var: Var };  
impl BitAnd for Boolean {  
    fn and(self: Boolean, Other: Boolean) -> Boolean {  
        //come prima  
        Boolean [ var: result ]  
    } }  
}
```



```
let a = Boolean::new_witness(|| true);  
let b = Boolean::new_witness(|| false);  
(a & b).enforce_equal(Boolean::FALSE);
```



::: Programmazione ZKP (13/20)

Ci sono molte librerie da impiegare per la definizione di sistemi di vincolo per i vari linguaggi di programmazione di alto livello, usando astrazioni di linguaggio:

- libsnark: gadgetlib per C++;
- arkworks: r1cs-std + crypto-primitive per Rust;
- Snarky per Ocaml;
- Gnark per Go.

Dato che si impiega un linguaggi di alto livello è possibile implementare computazione arbitrarie per generare testimoni e parametri per sistemi di prova.

::: Programmazione ZKP (13/20)

Ci sono molte librerie da impiegare per la definizione di sistemi di vincolo per i vari linguaggi di programmazione di alto livello, usando astrazioni di linguaggio:

- libsnark: gadgetlib per C++;
- arkworks: r1cs-std + crypto-primitive per Rust;
- Snarky per Ocaml;
- Gnark per Go.

Dato che si impiega un linguaggi di alto livello è possibile implementare computazione arbitrarie per generare testimoni e parametri per sistemi di prova.

Utilizzando la libreria arkworks, vediamo l'esempio del gioco del Sudoku definendo i due tipi per lo statement x e la soluzione w che sono definiti su un determinato campo finito e con una determinata dimensione N:

```
use ark_relations::r1cs::{SynthesisError, ConstraintSystem};
use cmp::CmpGadget;

mod cmp;
mod alloc;

pub struct Puzzle<const N: usize, ConstraintF: PrimeField>([[UInt8<ConstraintF>; N]; N]);
pub struct Solution<const N: usize, ConstraintF: PrimeField>([[UInt8<ConstraintF>; N];
N]);
```

::: Programmazione ZKP (14/20)

```
fn check_rows<const N: usize, ConstraintF: PrimeField>(
    solution: &Solution<N, ConstraintF>,
) -> Result<(), SynthesisError> {
    for row in &solution.0 {
        for (j, cell) in row.iter().enumerate() {
            for prior_cell in &row[0..j] {
                cell.is_neq(&prior_cell)?
                    .enforce_equal(&Boolean::TRUE)?;
            }
        }
    }
    Ok(())
}
```

Funzione Rust per il controllo che i lavori lungo una riga siano distinti.

::: Programmazione ZKP (14/20)

```
fn check_rows<const N: usize, ConstraintF: PrimeField>(
    solution: &Solution<N, ConstraintF>,
) -> Result<(), SynthesisError> {
    for row in &solution.0 {
        for (j, cell) in row.iter().enumerate() {
            for prior_cell in &row[0..j] {
                cell.is_neq(&prior_cell)?
                    .enforce_()
            }
        }
    }
    Ok(())
}
```

Funzione Rust per il controllo che la soluzione abbia gli stessi valori nelle celle valorizzate in puzzle.

Funzione Rust per il controllo che i lavori lungo una riga siano distinti.

```
fn check_puzzle_matches_solution<const N: usize, ConstraintF: PrimeField>(
    puzzle: &Puzzle<N, ConstraintF>,
    solution: &Solution<N, ConstraintF>,
) -> Result<(), SynthesisError> {
    for (p_row, s_row) in puzzle.0.iter().zip(&solution.0) {
        for (p, s) in p_row.iter().zip(s_row) {
            // Ensure that the solution `s` is in the range [1, N]
            s.is_leq(&UInt8::constant(N as u8))?
                .and(&s.is_geq(&UInt8::constant(1))?)?
                .enforce_equal(&Boolean::TRUE)?;

            // Ensure that either the puzzle slot is 0, or that
            // the slot matches equivalent slot in the solution
            (p.is_eq(s)?.or(&p.is_eq(&UInt8::constant(0))?)?).
                .enforce_equal(&Boolean::TRUE)?;
        }
    }
    Ok(())
}
```

::: Programmazione ZKP (15/20)

```
fn check_helper<const N: usize, ConstraintF: PrimeField>(
    puzzle: &[[u8; N]; N],
    solution: &[[u8; N]; N],
) {
    let cs = ConstraintSystem::<ConstraintF>::new_ref();
    let puzzle_var = Puzzle::new_input(cs.clone(), || Ok(puzzle)).unwrap();
    let solution_var = Solution::new_witness(cs.clone(), || Ok(solution)).unwrap();
    check_puzzle_matches_solution(&puzzle_var, &solution_var).unwrap();
    check_rows(&solution_var).unwrap();
    assert!(cs.is_satisfied().unwrap());
}
```

Funzione Rust di supporto per l'esecuzione di tutti i controlli sui due oggetti di input.

::: Programmazione ZKP (15/20)

```
fn check_helper<const N: usize, ConstraintF: PrimeField>(
    puzzle: &[[u8; N]; N],
    solution: &[[u8; N]; N],
) {
    let cs = ConstraintSystem::<ConstraintF>::new_ref();
    let puzzle_var = Puzzle::new_input(cs.clone(), || Ok(puzzle)).unwrap();
    let solution_var = Solution::new_witness(cs.clone(), || Ok(solution)).unwrap();
    check_puzzle_matches_solution(&puzzle_var, &solution_var).unwrap();
    check_rows(&solution_var).unwrap();
    assert!(cs.is_satisfied().unwrap());
}
```

Funzione Rust di supporto per l'esecuzione di tutti i controlli sui due oggetti di input.

```
fn main() {
    use ark_bls12_381::Fq as F;
    // Check that it accepts a valid solution.
    let puzzle = [
        [1, 0],
        [0, 2],
    ];
    let solution = [
        [1, 2],
        [1, 2],
    ];
    check_helper::<2, F>(&puzzle, &solution);
```

Istanziazione di due esempi di x e w e controllo del rispetto dei vincoli su queste istanze.

::: Programmazione ZKP (15/20)

```
fn check_helper<const N: usize, ConstraintF: PrimeField>(
    puzzle: &[[u8; N]; N],
    solution: &[[u8; N]; N],
) {
    let cs = ConstraintSystem::<ConstraintF>::new_ref();
    let puzzle_var = Puzzle::new_input(cs.clone(), || Ok(puzzle)).unwrap();
    let solution_var = Solution::new_witness(cs.clone(), || Ok(solution)).unwrap();
    check_puzzle_matches_solution(&puzzle_var, &solution_var).unwrap();
    check_rows(&solution_var).unwrap();
    assert!(cs.is_satisfied().unwrap());
}
```

Funzione Rust di supporto per l'esecuzione di tutti i controlli sui due oggetti di input.

Eseguendo cargo run, possiamo compilare ed eseguire il programma che termina con successo. Variando la soluzione con [1, 0; 1, 2] si ottiene l'avviso di violazione di un vincolo. Successivamente è possibile usare questo su ogni sistema di prova come Groth'16

```
fn main() {
    use ark_bls12_381::Fq as F;
    // Check that it accepts a valid solution.
    let puzzle = [
        [1, 0],
        [0, 2],
    ];
    let solution = [
        [1, 2],
        [1, 2],
    ];
    check_helper::<2, F>(&puzzle, &solution);
```

Istanziazione di due esempi di x e w e controllo del rispetto dei vincoli su queste istanze.

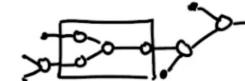
::: Programmazione ZKP (16/20)

Program

```
fn main(...){  
    ...  
}
```

Compiler

R1CS



Dato un programma di verifica scritto in un linguaggio di alto livello, si vuole ottenere un artefatto scritto in un formalismo di basso livello come R1CS. ZoKrates è uno dei framework più noti per effettuare questo.

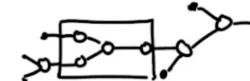
::: Programmazione ZKP (16/20)

Program

```
fn main(...){  
    ...  
}
```

Compiler

R1CS



Dato un programma di verifica scritto in un linguaggio di alto livello, si vuole ottenere un artefatto scritto in un formalismo di basso livello come R1CS. ZoKrates è uno dei framework più noti per effettuare questo.

Supporta la definizione di tipi strutturati o
di alias di tipo

Ogni programma ZoKrates ha un main

type F = field;

def main(public F x, private F[2] ys) {

}

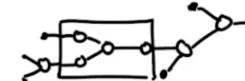
::: Programmazione ZKP (16/20)

Program

```
fn main(...){  
    ...  
}
```

Compiler

R1CS



Dato un programma di verifica scritto in un linguaggio di alto livello, si vuole ottenere un artefatto scritto in un formalismo di basso livello come R1CS. ZoKrates è uno dei framework più noti per effettuare questo.

Supporta la definizione di tipi strutturati o
di alias di tipo

Ogni programma ZoKrates ha un main,
dove definire nuove variabili da usare
durante l'esecuzione o la dimostrazione,
e verificare vincoli mediante la parola
chiave «assert».

Questo metodo è differente rispetto a
quanto visto con Artworks: possono
prendere ogni espressione booleana come
input, e non c'è bisogno di alcuna
combinazione lineare.

```
type F = field;  
  
def main(public F x, private F[2] ys) {  
    field y0 = y[0];  
    field y1 = y[1];  
    assert(x = y0 * y1);  
}
```

::: Programmazione ZKP (17/20)

Supporta la programmazione generica con interi per le funzioni

```
def repeated_squaring<N>(field x) ->
    field {
        }
```

::: Programmazione ZKP (17/20)

Supporta la programmazione generica con interi per le funzioni, per la definizione ed impiego di array (che possono essere mutati), e l'uso di cicli.

```
def repeated_squaring<N>(field x) ->
    field {
        field[N] mut xs;
        xs[0] = x;
        for u32 i in 0..N {
            xs[i + 1] = xs[i] * xs[i];
        }
        return xs[N];
    }
```

::: Programmazione ZKP (17/20)

Supporta la programmazione generica con interi per le funzioni, per la definizione ed impiego di array (che possono essere mutati), e l'uso di cicli.

Bisogna sempre specificare un main per l'uso della funzione.

```
def repeated_squaring<N>(field x) ->
    field {
        field[N] mut xs;
        xs[0] = x;
        for u32 i in 0..N {
            xs[i + 1] = xs[i] * xs[i];
        }
        return xs[N];
    }
```

```
def main (public field x) -> field {
    repeated_squaring::<1000>(x)
}
```

::: Programmazione ZKP (17/20)

Supporta la programmazione generica con interi per le funzioni, per la definizione ed impiego di array (che possono essere mutati), e l'uso di cicli.

Bisogna sempre specificare un main per l'uso della funzione.

Non c'è nessun modo in ZoKrates di computare il testimone w , ma tutti i testimoni devono essere forniti come input alla funzione main. Non è possibile dichiarare una variabile provata da impiegare durante l'esecuzione.

```
def repeated_squaring<N>(field x) ->
    field {
        field[N] mut xs;
        xs[0] = x;
        for u32 i in 0..N {
            xs[i + 1] = xs[i] * xs[i];
        }
        return xs[N];
    }
```

```
def main (public field x) -> field {
    repeated_squaring::<1000>(x)
}
```

::: Programmazione ZKP (18/20)

Si implementi l'esempio del Sudoku in ZoKrates:

```
struct Puzzle<N> {
    u8[N][N] elems;
}
struct Solution<N> {
    u8[N][N] elems;
}
```

Vengono definite i due tipi per la definizione dello statement x e del testimone w.

::: Programmazione ZKP (18/20)

Si implementi l'esempio del Sudoku in ZoKrates:

```
struct Puzzle<N> {
    u8[N][N] elems;
}
struct Solution<N> {
    u8[N][N] elems;
}
```

Vengono definite i due tipi per la definizione dello statement x e del testimone w.

Funzione per la verifica che tutti gli elementi su una riga della soluzione siano distinti.

```
def check_rows<N>(Solution<N> sol) -> bool {
    // for each row
    for u32 i in 0..N {
        // for each entry in each row
        for u32 j in 0..N {
            // check that (i, j)-th element is not equal to any of the
            // previous elements in that row
            for u32 k in 0..j {
                assert(sol.elems[i][j] != sol.elems[i][k]);
            }
        }
    }
    return true;
}
```

::: Programmazione ZKP (19/20)

```
def check_puzzle_matches_solution<N>(Solution<N> sol, Puzzle<N> puzzle) -> bool {
    for u32 i in 0..N {
        for u32 j in 0..N {
            assert((sol.elems[i][j] > 0) && (sol.elems[i][j] < 10));
            assert((puzzle.elems[i][j] == 0) || (puzzle.elems[i][j] == sol.elems[i][j]));
        }
    }
    return true;
}
```

Funzione per il controllo che gli elementi nel puzzle e nella soluzioni corrispondono se valorizzati nel primo.

::: Programmazione ZKP (19/20)

```
def check_puzzle_matches_solution<N>(Solution<N> sol, Puzzle<N> puzzle) -> bool {
    for u32 i in 0..N {
        for u32 j in 0..N {
            assert((sol.elems[i][j] > 0) && (sol.elems[i][j] < 10));
            assert((puzzle.elems[i][j] == 0) || (puzzle.elems[i][j] == sol.elems[i][j]));
        }
    }
    return true;
}
```

Funzione per il controllo che gli elementi nel puzzle e nella soluzioni corrispondono se valorizzati nel primo.

```
def main(public Puzzle<2> puzzle, private Solution<2> sol) {
    assert(check_puzzle_matches_solution(sol, puzzle));
    assert(check_rows(sol));
}
```

Funzione main che prende in ingresso puzzle e soluzione e avvia i controlli.

::: Programmazione ZKP (19/20)

```
def check_puzzle_matches_solution<N>(Solution<N> sol, Puzzle<N> puzzle) -> bool {
    for u32 i in 0..N {
        for u32 j in 0..N {
            assert((sol.elems[i][j] > 0) && (sol.elems[i][j] < 10));
            assert((puzzle.elems[i][j] == 0) || (puzzle.elems[i][j] == sol.elems[i][j]));
        }
    }
    return true;
}
```

Funzione per il controllo che gli elementi nel puzzle e nella soluzioni corrispondono se valorizzati nel primo.

```
def main(public Puzzle<2> puzzle, private Solution<2> sol) {
    assert(check_puzzle_matches_solution(sol, puzzle));
    assert(check_rows(sol));
}
```

Funzione main che prende in ingresso puzzle e soluzione e avvia i controlli.

A questo punto è possibile compilare il programma:

\$ zokrates compile –input sudoku.zok

generare il setup con il sistema di default che è Groth'16:

\$ zokrates setup

ottenere il testimone: \$zokrates compute-witness –a **1 0 0 2 | 1 2 1 2**

verificare lo schema di prova con x e w con \$ zokrates verify

Statement x, puzzle da risolvere.

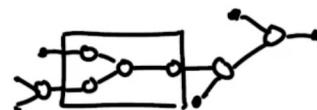
Testimone w, soluzione al puzzle.

::: Programmazione ZKP (20/20)

Sono possibili tre diversi approcci per la programmazione di sistemi di prova ZK:

HDL

Un linguaggio
per descrivere la
sintesi di circuiti



Library

Una libreria per descrivere
la sintesi di circuiti

```
Circ.add-wire(...)  
circ.add-gate(...)
```

PL + Compiler

Un linguaggio compilato
per descrivere la sintesi di
circuiti

```
fn main(...){  
...  
}
```

Linguaggio autonomo?

Ognuna di queste soluzioni ha i propri pro e contro.

- Circom presenta una sintassi elegante e chiara per definire i vincoli.
- È difficile da imparare perché non è un linguaggio di programmazione ed è limitato per creare delle buone astrazioni (no tipi).

Tipo di
Linguaggio

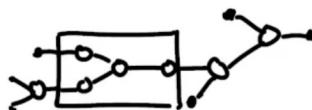
	No	Yes
Circuit	Library (arkworks)	HDL (circom)
Program		PL (ZoKrates)

::: Programmazione ZKP (20/20)

Sono possibili tre diversi approcci per la programmazione di sistemi di prova ZK:

HDL

Un linguaggio
per descrivere la
sintesi di circuiti



Library

Una libreria per descrivere
la sintesi di circuiti

`Circ.add-wire(...)`
`Circ.add-gate(...)`

PL + Compiler

Un linguaggio compilato
per descrivere la sintesi di
circuiti

```
fn main(...){  
    ...  
}
```

Linguaggio autonomo?

Ognuna di queste soluzioni ha i propri pro e contro.

- Arkworks consente la chiara descrizione di vincoli di Circom, in aggiunta ha l'espressività di Rust.
- È difficile da imparare se non si conosce il linguaggio host, ovvero Rust. Offre poche ottimizzazioni, per minimizzare i vincoli.

Tipo di
Linguaggio

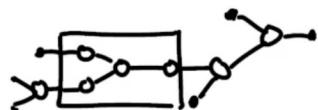
	No	Yes
Circuit	Library (arkworks)	HDL (circom)
Program		PL (ZoKrates)

::: Programmazione ZKP (20/20)

Sono possibili tre diversi approcci per la programmazione di sistemi di prova ZK:

HDL

Un linguaggio
per descrivere la
sintesi di circuiti



Library

Una libreria per descrivere
la sintesi di circuiti

```
Circ.add-wire(...)  
circ.add-gate(...)
```

PL + Compiler

Un linguaggio compilato
per descrivere la sintesi di
circuiti

```
fn main(...){  
...  
}
```

Linguaggio autonomo?

Ognuna di queste soluzioni ha i propri pro e contro.

- ZoKrates è facile da imparare e presenta un'elegante sintassi.
- È limitato nella generazione del testimone.

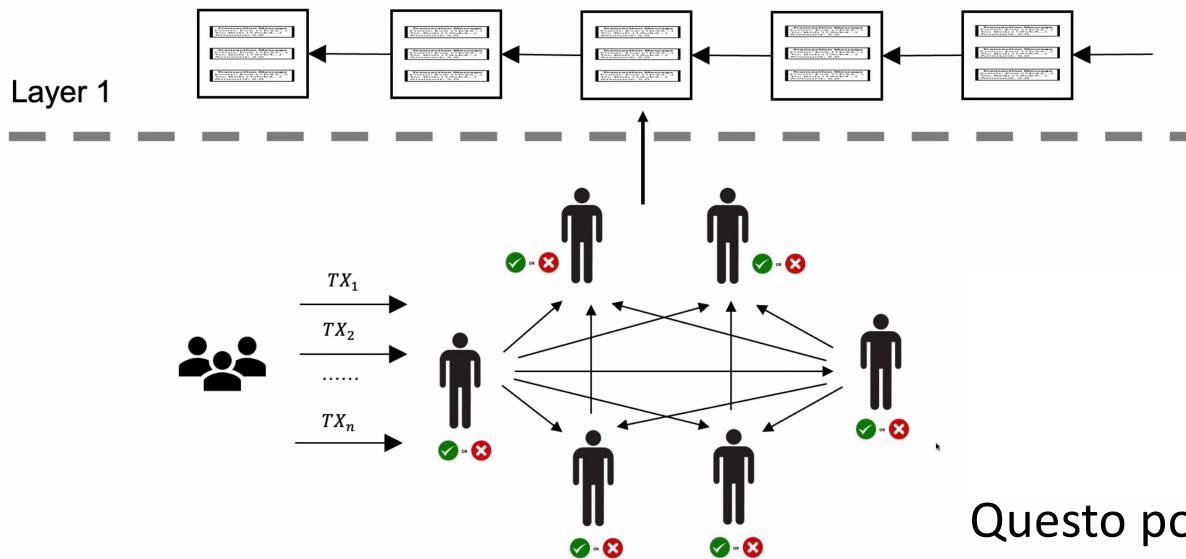
Tipo di
Linguaggio

	No	Yes
Circuit	Library (arkworks)	HDL (circom)
Program		PL (ZoKrates)



Applicazioni nelle Blockchain

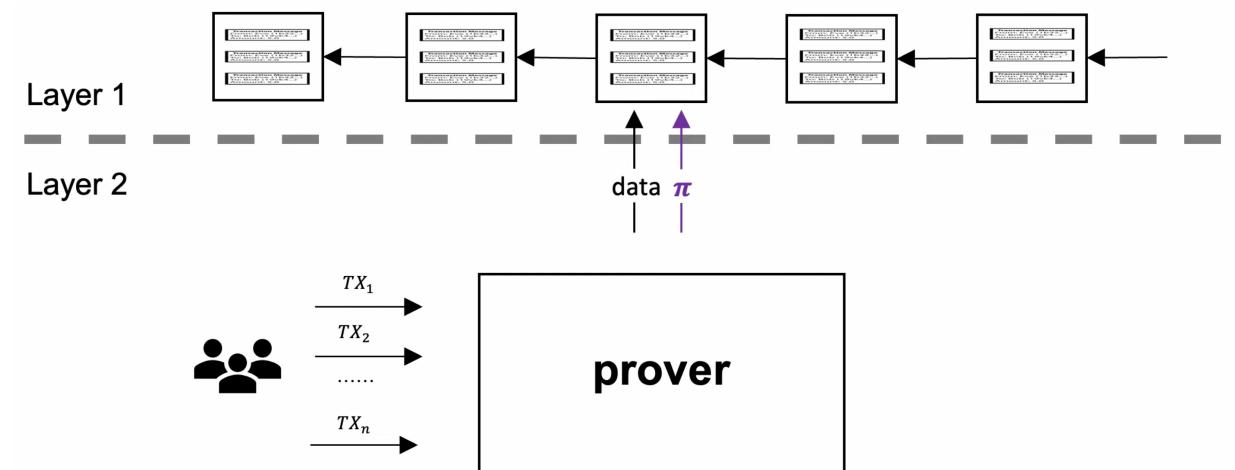
... ZK-Rollup in Blockchain (1/7)



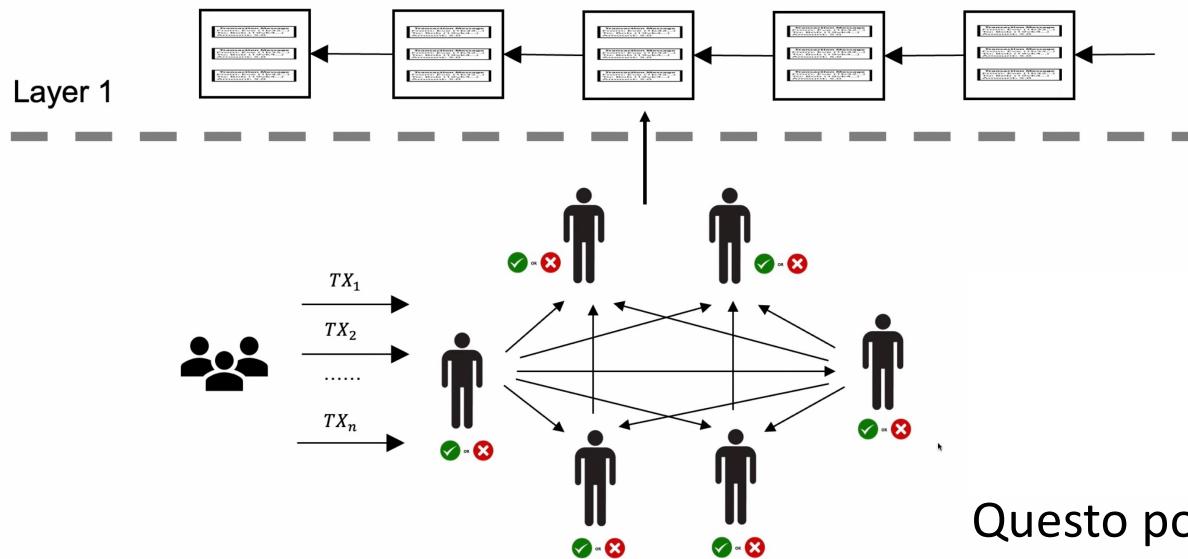
Soluzioni a questo problema sono servizi di livello 2 con un sottoinsieme di nodi e un approccio più centralizzato per la raccolta, validazione ed approvazione delle transazioni, con ribaltamento sulla catena principale dei dati di aggiornamento di stato e di una prova di correttezza ZK non-interattiva.

Una blockchain rappresenta un sistema che richiede l'esecuzione di un complesso ed oneroso algoritmo di consenso distribuito per la validazione dei blocchi che includono transazioni sottomessi da utenti.

Questo pone limiti sul throughput, scalabilità e prestazioni della soluzione blockchain.



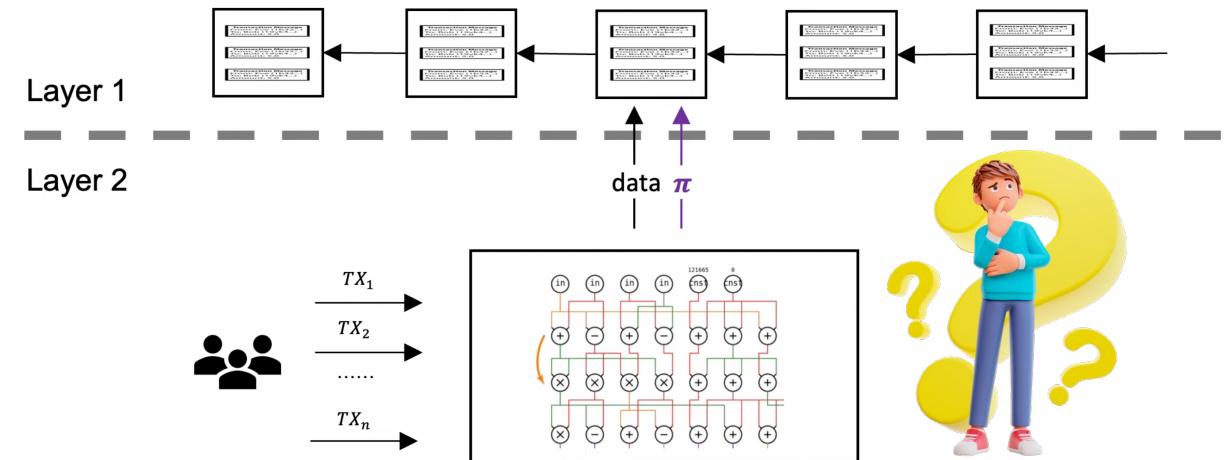
... ZK-Rollup in Blockchain (1/7)



Soluzioni a questo problema sono servizi di livello 2 con un sottoinsieme di nodi e un approccio più centralizzato per la raccolta, validazione ed approvazione delle transazioni, con ribaltamento sulla catena principale dei dati di aggiornamento di stato e di una prova di correttezza ZK non-interattiva.

Una blockchain rappresenta un sistema che richiede l'esecuzione di un complesso ed oneroso algoritmo di consenso distribuito per la validazione dei blocchi che includono transazioni sottomessi da utenti.

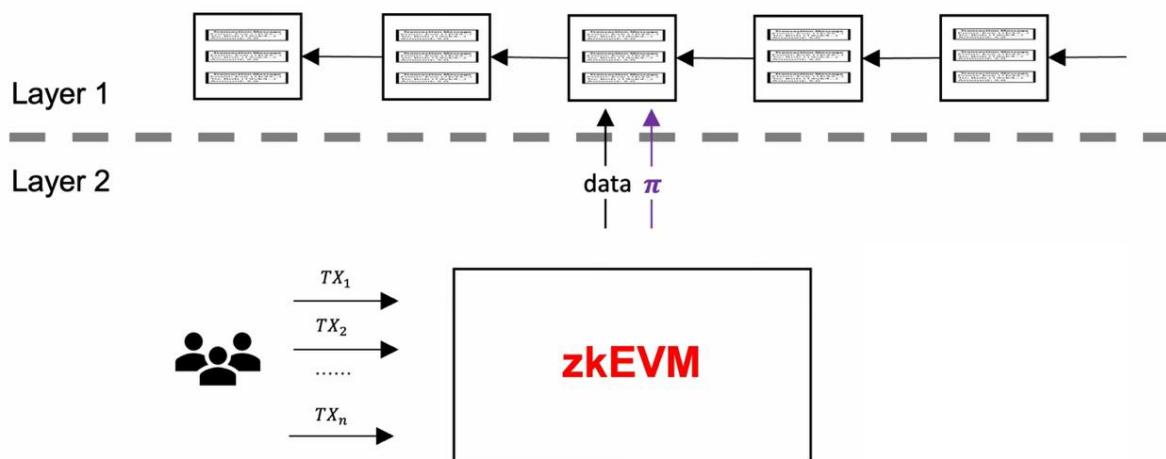
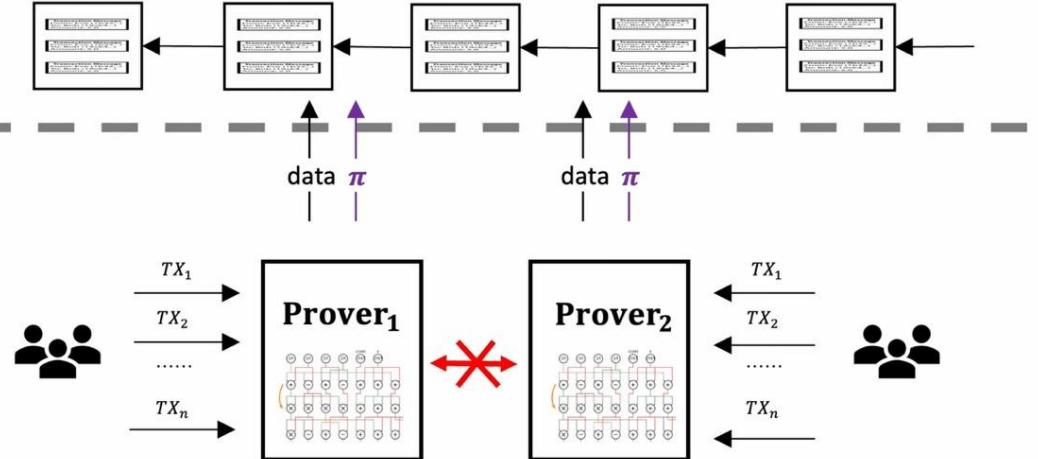
Questo pone limiti sul throughput, scalabilità e prestazioni della soluzione blockchain.



Come implementare il prover di transazioni con un circuito aritmetico? Ciò è richiesto anche a un developer Solidity per la validazione di smart contracts.

... ZK-Rollup in Blockchain (2/7)

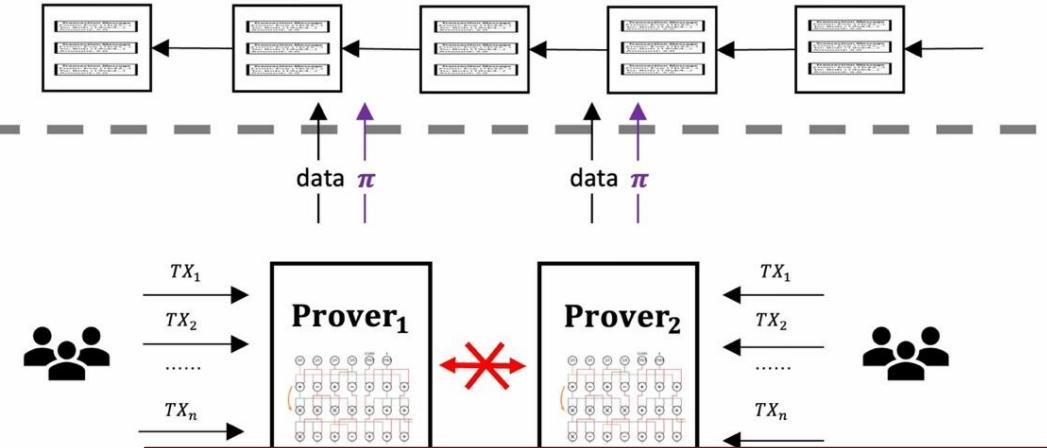
Anche nel caso di riuscire ad implementare un prover (con un insieme di circuiti) per una data applicazione/smart contract A e per un altro per un altro tipo di applicazione/smart contract B, questi non sono componibili ed integrabili per una prova atomica di più interazioni.



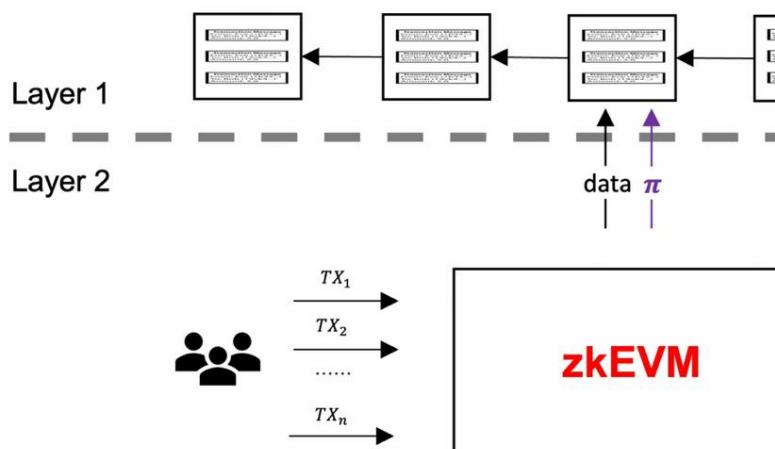
L'obiettivo è quello di evitare che un developer Solidity debba implementare circuiti e di avere componibilità delle prove. L'idea di base è di non effettuare prove per la logica specifica di un'applicazione, ma di costruire un sistema di prova a livello di EVM con uno stato globale condiviso per esecuzione corretta di transazioni.

::: ZK-Rollup in Blockchain (2/7)

Anche nel caso di riuscire ad implementare un prover (con un insieme di circuiti) per una data applicazione/smart contract A e per un altro per un altro tipo di applicazione/smart contract B, questi non sono componibili ed integrabili per una prova atomica di più interazioni.



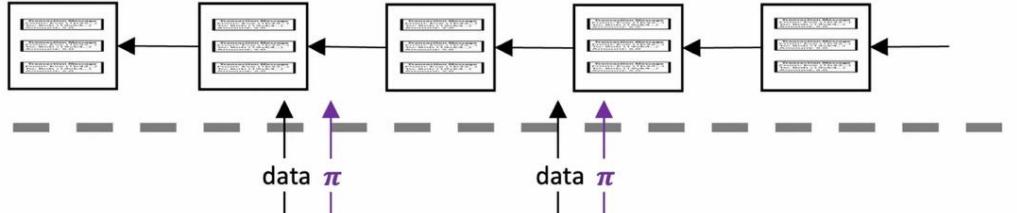
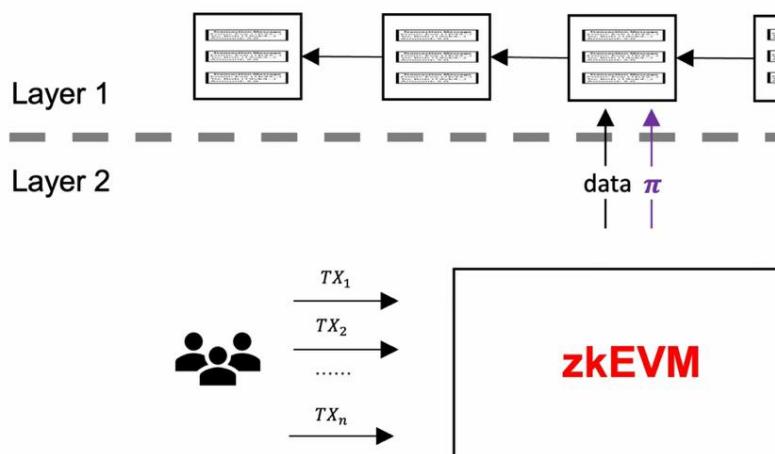
Questa idea è complessa da implementare dovendo considerare molti opcode a livello EVM. Inoltre, le prove per n transazioni possono avere un enorme tempo di generazione.



Una delle idee è di non effettuare prove per la logica specifica di un'applicazione, ma di costruire un sistema di prova a livello di EVM con uno stato globale condiviso per esecuzione corretta di transazioni.

::: ZK-Rollup in Blockchain (2/7)

Anche nel caso di riuscire ad implementare un prover (con un insieme di circuiti) per una data applicazione/smart contract A e per un altro per un altro tipo di applicazione/smart contract B, questi non sono componibili ed integrabili per una prova atomica di più interazioni.

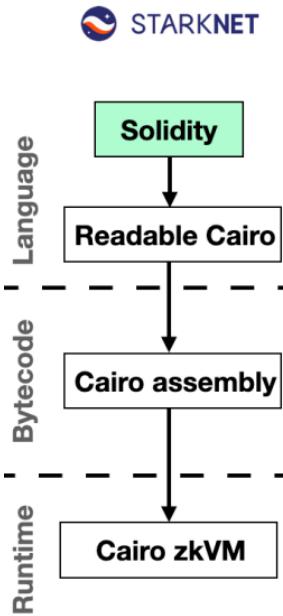


Recenti sviluppi dei sistemi di prova ZK negli ultimi anni hanno fornito soluzioni come commitment polinomiale, hardware acceleration e prove ricorsive per aumentare di tre volte l'efficienza dei sistemi di prova e per rendere possibile l'idea della zkEVM, che è diventata popolare.

Recenti sviluppi dei sistemi di prova ZK negli ultimi anni hanno fornito soluzioni come commitment polinomiale, hardware acceleration e prove ricorsive per aumentare di tre volte l'efficienza dei sistemi di prova e per rendere possibile l'idea della zkEVM, che è diventata popolare. L'idea di base è di non effettuare prove per la logica specifica di un'applicazione, ma di costruire un sistema di prova a livello di EVM con uno stato globale condiviso per esecuzione corretta di transazioni.

::: ZK-Rollup in Blockchain (3/7)

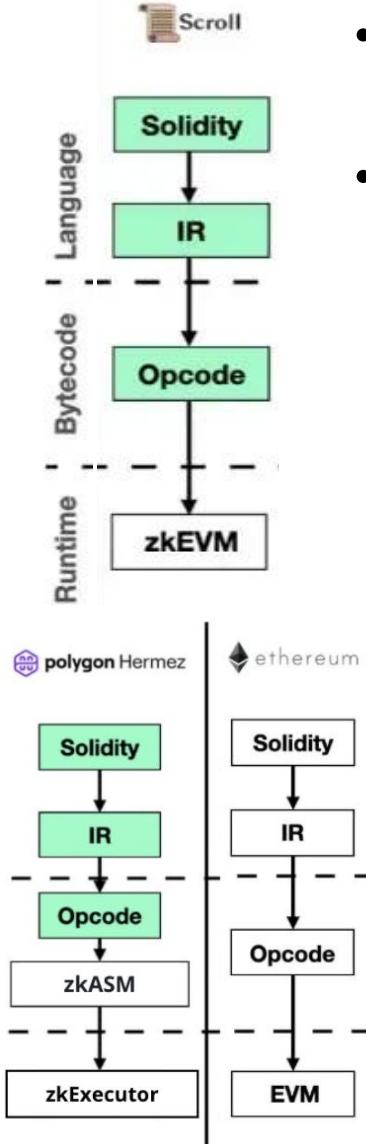
Ci sono una serie di miglioramenti per l'impiego di zkEVM:



- EVM non è stata sviluppata per integrare sistemi di prova ZK, e la scrittura di circuiti per EVM comporta un grande overhead per la quantità di opcode che sono non ottimale (o anche di ostacolo) per zkEVM. Si è pensato di riprogettare e di realizzare una nuova VM che sia più amichevole per zkEVM. Questo comporta un problema di compatibilità e si richiede un compilatore per convertire artifatti in linguaggi supportati da EVM (come Solidity) per ottenere del bytecode compatibile con la nuova EVM. Questa language-level zkEVM è la soluzione di Starkware con l'infrastruttura StarkNet e la Cairo zkVM.

... ZK-Rollup in Blockchain (3/7)

Ci sono una serie di possibili soluzioni per l'impiego di zkEVM:

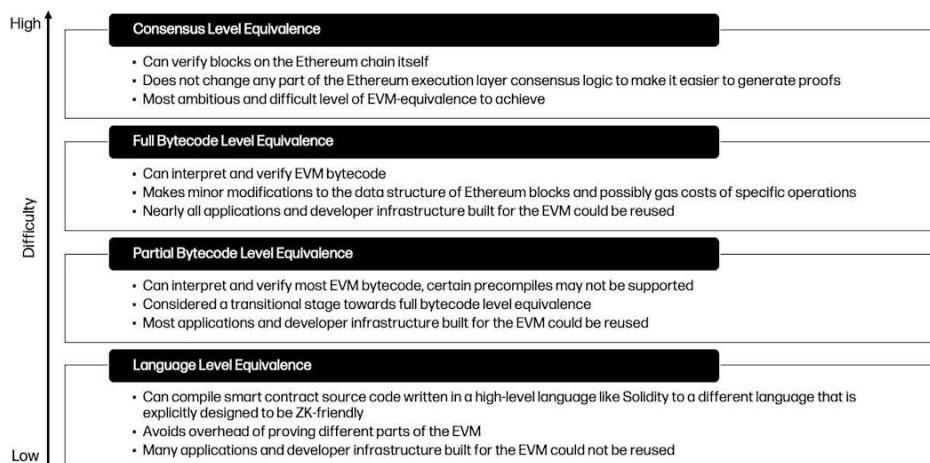


- Questa language-level zkEVM è la soluzione di Starkware con l'infrastruttura StarkNet e la Cairo zkVM.
- La seconda è una bytecode-level zkEVM dove si ottiene il bytecode dalla compilazione di codice Solidity (o altro) e si prova che il bytecode è stato eseguito correttamente. La parte di esecuzione è ancora EVM ma si possono introdurre ottimizzazioni di come è implementata la logica applicativa e/o le strutture dati per renderle più ottimizzate ad una prova zkEVM. Si può incappare in tempi più lunghi di generazione delle prove perché ci possono essere primitive non amichevoli. Scroll converte l'esecuzione EVM direttamente in circuiti verificabili, ma non supporta tutti gli opcode, mentre Polygon Hermez esegue una VM personalizzata ma ha mappato tutti gli opcode EVM in opcode validi per la propria VM. Rispetto a scroll, Polygon ha un runtime alternativo ("zkExecutor") che esegue opcode "zkASM" personalizzati anziché opcode EVM direttamente.

::: ZK-Rollup in Blockchain (3/7)

Ci sono una serie di possibili soluzioni per l'impiego di zkEVM:

- Questa language-level zkEVM è la soluzione di Starkware con l'infrastruttura StarkNet e la Cairo zkVM.
- La seconda è una bytecode-level zkEVM come Scroll o Polygon Hermez.
- L'ultimo insieme di soluzioni è detta a livello di consenso, perché lascia invariata l'infrastruttura Ethereum (come la EVM, lo storage e altre informazioni nell'header del blocco). Ha però maggiori tempi di prova e potrebbe essere vulnerabile a potenziali attacchi DDoS. Non sussistono soluzioni disponibili attualmente.



Level of EVM Equivalence	ZK Proving Algorithm	Volition	Smart Contract Language	Open Sourced	Expected Mainnet Launch	
zkSync 2.0	Language	SNARK	Yes	Solidity Zinc Compiler	No	EOY 2022
StarkNet + Warp	Language	STARK	Yes	Solidity Cairo Compiler	No	Already available*
Polygon zkEVM	Bytecode (Partial)	SNARK+STARK	TBD	Solidity	Yes**	H1 2023
Scroll	Bytecode (Partial)	SNARK	No	Solidity	Yes	TBD
Privacy Scaling Explorations (Ethereum Foundation)	Consensus	SNARK	TBD	Solidity	Yes	TBD

*Several features of Solidity are not supported in StarkNet.

**Polygon zkEVM specifications are public but not published under an open-source license.

::: ZK-Rollup in Blockchain (4/7)

L'impiego di una soluzione di prova implica la scrittura di un programma, la sua traduzione in uno dei possibili formati di rappresentazione e l'impiego di schemi di commitment e IOP per la generazione della prova.

```
def hcf(x, y):
    if x > y:
        smaller = y
    else:
        smaller = x

    for i in range(1,smaller + 1):
        if((x % i == 0) and (y % i == 0)):
            hcf = i

    return hcf
```



R1CS
Plonkish
AIR

$$\begin{aligned} x * x &== \text{var1} \\ \text{var1} * x &== y \\ (y+x) * 1 &== \text{var2} \\ (\text{var2}+5) * 1 &== \text{out} \end{aligned}$$

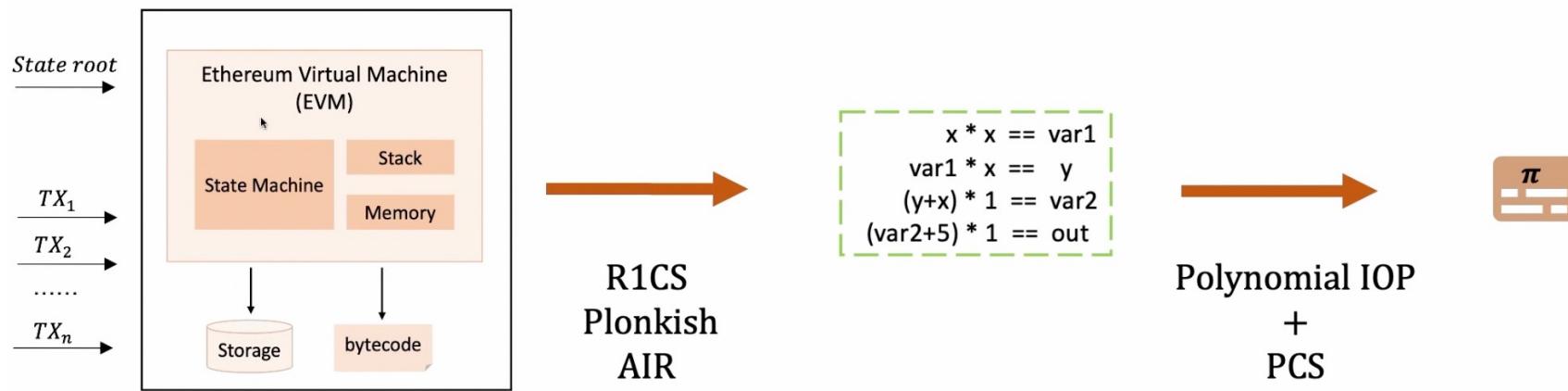


Polynomial IOP
+
PCS

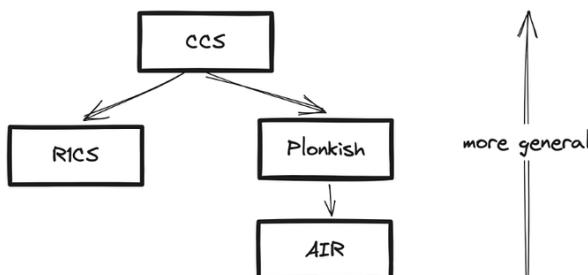


::: ZK-Rollup in Blockchain (4/7)

L'impiego di una soluzione di prova implica la scrittura di un programma, la sua traduzione in uno dei possibili formati di rappresentazione e l'impiego di schemi di commitment e IOP per la generazione della prova.



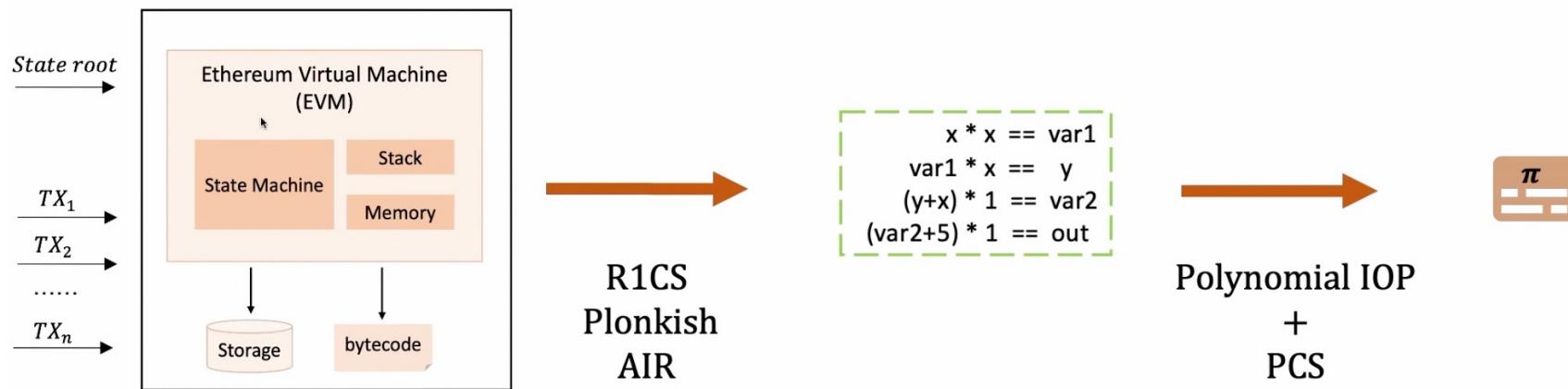
Siccome si deve provare l'esecuzione su EVM, quindi il programma è la stessa EVM. La soluzione più usata è scrivere i vincoli usando Plonkish e usare Plonk IOP e lo schema di commitment polinomiale KZG nel backend per la generazione della prova.



Plonkish è più flessibile e articolato di R1CS che ha una struttura più fissa, e ha una sintassi in termini di gate più articolata che consente una maggiore granularità ed espressività nella scrittura dei vincoli.

.... ZK-Rollup in Blockchain (4/7)

L'impiego di una soluzione di prova implica la scrittura di un programma, la sua traduzione in uno dei possibili formati di rappresentazione e l'impiego di schemi di commitment e IOP per la generazione della prova.



Siccome si deve provare l'esecuzione su EVM, quindi il programma è la stessa EVM. La soluzione più usata è scrivere i vincoli usando Plonkish e usare Plonk IOP e lo schema di commitment polinomiale KZG nel backend per la generazione della prova.

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$							
va_1	vb_1	vc_1							
va_2	vb_2	vc_2			vd_2				
va_3	vb_3	vc_3			vd_3				
va_4	vb_4	vc_4		vd_4				
va_5	vb_5	vc_5			vd_5				
va_6	vb_6	vc_6			vd_6				
va_7	vb_7	vc_7			vd_7				
va_8	vb_8	vc_8			vd_8				
va_9	vb_9	vc_9			vd_9				
va_{10}	vb_{10}	vc_{10}			vd_{10}				
va_{11}	vb_{11}	vc_{11}			vd_{11}				
va_{12}	vb_{12}	vc_{12}			vd_{12}				
va_{13}	vb_{13}	vc_{13}			vd_{13}				
va_{14}	vb_{14}	vc_{14}			vd_{14}				
va_{15}	vb_{15}	vc_{15}			vd_{15}				
va_{16}	vb_{16}	vc_{16}			vd_{16}				
va_{17}	vb_{17}	vc_{17}			vd_{17}				
va_{18}	vb_{18}	vc_{18}			vd_{18}				
va_{19}	vb_{19}	vc_{19}			vd_{19}				
va_{20}	vb_{20}	vc_{20}			vd_{20}				
va_{21}	vb_{21}	vc_{21}			vd_{21}				
va_{22}	vb_{22}	vc_{22}			vd_{22}				
va_{23}	vb_{23}	vc_{23}			vd_{23}				
va_{24}	vb_{24}	vc_{24}			vd_{24}				
va_{25}	vb_{25}	vc_{25}			vd_{25}				
va_{26}	vb_{26}	vc_{26}			vd_{26}				
va_{27}	vb_{27}	vc_{27}			vd_{27}				
va_{28}	vb_{28}	vc_{28}			vd_{28}				
va_{29}	vb_{29}	vc_{29}			vd_{29}				
va_{30}	vb_{30}	vc_{30}			vd_{30}				
va_{31}	vb_{31}	vc_{31}			vd_{31}				
va_{32}	vb_{32}	vc_{32}			vd_{32}				
va_{33}	vb_{33}	vc_{33}			vd_{33}				
va_{34}	vb_{34}	vc_{34}			vd_{34}				
va_{35}	vb_{35}	vc_{35}			vd_{35}				
va_{36}	vb_{36}	vc_{36}			vd_{36}				
va_{37}	vb_{37}	vc_{37}			vd_{37}				
va_{38}	vb_{38}	vc_{38}			vd_{38}				
va_{39}	vb_{39}	vc_{39}			vd_{39}				
va_{40}	vb_{40}	vc_{40}			vd_{40}				
va_{41}	vb_{41}	vc_{41}			vd_{41}				
va_{42}	vb_{42}	vc_{42}			vd_{42}				
va_{43}	vb_{43}	vc_{43}			vd_{43}				
va_{44}	vb_{44}	vc_{44}			vd_{44}				
va_{45}	vb_{45}	vc_{45}			vd_{45}				
va_{46}	vb_{46}	vc_{46}			vd_{46}				
va_{47}	vb_{47}	vc_{47}			vd_{47}				
va_{48}	vb_{48}	vc_{48}			vd_{48}				
va_{49}	vb_{49}	vc_{49}			vd_{49}				
va_{50}	vb_{50}	vc_{50}			vd_{50}				
va_{51}	vb_{51}	vc_{51}			vd_{51}				
va_{52}	vb_{52}	vc_{52}			vd_{52}				
va_{53}	vb_{53}	vc_{53}			vd_{53}				
va_{54}	vb_{54}	vc_{54}			vd_{54}				
va_{55}	vb_{55}	vc_{55}			vd_{55}				
va_{56}	vb_{56}	vc_{56}			vd_{56}				
va_{57}	vb_{57}	vc_{57}			vd_{57}				
va_{58}	vb_{58}	vc_{58}			vd_{58}				
va_{59}	vb_{59}	vc_{59}			vd_{59}				
va_{60}	vb_{60}	vc_{60}			vd_{60}				
va_{61}	vb_{61}	vc_{61}			vd_{61}				
va_{62}	vb_{62}	vc_{62}			vd_{62}				
va_{63}	vb_{63}	vc_{63}			vd_{63}				
va_{64}	vb_{64}	vc_{64}			vd_{64}				
va_{65}	vb_{65}	vc_{65}			vd_{65}				
va_{66}	vb_{66}	vc_{66}			vd_{66}				
va_{67}	vb_{67}	vc_{67}			vd_{67}				
va_{68}	vb_{68}	vc_{68}			vd_{68}				
va_{69}	vb_{69}	vc_{69}			vd_{69}				
va_{70}	vb_{70}	vc_{70}			vd_{70}				
va_{71}	vb_{71}	vc_{71}			vd_{71}				
va_{72}	vb_{72}	vc_{72}			vd_{72}				
va_{73}	vb_{73}	vc_{73}			vd_{73}				
va_{74}	vb_{74}	vc_{74}			vd_{74}				
va_{75}	vb_{75}	vc_{75}			vd_{75}				
va_{76}	vb_{76}	vc_{76}			vd_{76}				
va_{77}	vb_{77}	vc_{77}			vd_{77}				
va_{78}	vb_{78}	vc_{78}			vd_{78}				
va_{79}	vb_{79}	vc_{79}			vd_{79}				
va_{80}	vb_{80}	vc_{80}			vd_{80}				
va_{81}	vb_{81}	vc_{81}			vd_{81}				
va_{82}	vb_{82}	vc_{82}			vd_{82}				
va_{83}	vb_{83}	vc_{83}			vd_{83}				
va_{84}	vb_{84}	vc_{84}			vd_{84}				
va_{85}	vb_{85}	vc_{85}			vd_{85}				
va_{86}	vb_{86}	vc_{86}			vd_{86}				
va_{87}	vb_{87}	vc_{87}			vd_{87}				
va_{88}	vb_{88}	vc_{88}			vd_{88}				
va_{89}	vb_{89}	vc_{89}			vd_{89}				
va_{90}	vb_{90}	vc_{90}			vd_{90}				
va_{91}	vb_{91}	vc_{91}			vd_{91}				
va_{92}	vb_{92}	vc_{92}			vd_{92}				
va_{93}	vb_{93}	vc_{93}			vd_{93}				
va_{94}	vb_{94}	vc_{94}			vd_{94}				
va_{95}	vb_{95}	vc_{95}			vd_{95}				
va_{96}	vb_{96}	vc_{96}			vd_{96}				
va_{97}	vb_{97}	vc_{97}			vd_{97}				
va_{98}	vb_{98}	vc_{98}			vd_{98}				
va_{99}	vb_{99}	vc_{99}			vd_{99}				

Table 1 Table 2

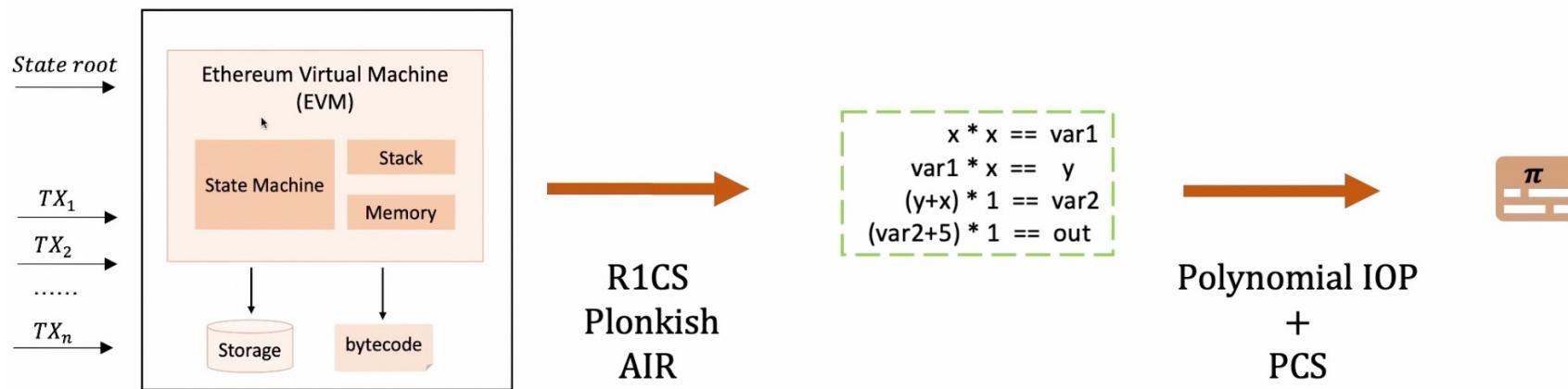
witness

$$\begin{aligned}
 &vb_1 * vc_1 + vc_2 - vc_3 = 0 \\
 &va_3 * vb_3 * vc_3 - vb_4 = 0 \\
 &vb_4 + vc_6 * vb_6 - va_6 = 0 \\
 &\dots \\
 &vb_4 = vc_6 = vb_6 = va_6 \\
 &\dots \\
 &(va_7, vb_7, vc_7) \in (T_0, T_1, T_2)
 \end{aligned}$$

Plonkish è più flessibile e articolato di R1CS che ha una struttura più fissa, e ha una sintassi in termini di gate più articolata che consente una maggiore granularità ed espressività nella scrittura dei vincoli.

::: ZK-Rollup in Blockchain (4/7)

L'impiego di una soluzione di prova implica la scrittura di un programma, la sua traduzione in uno dei possibili formati di rappresentazione e l'impiego di schemi di commitment e IOP per la generazione della prova.

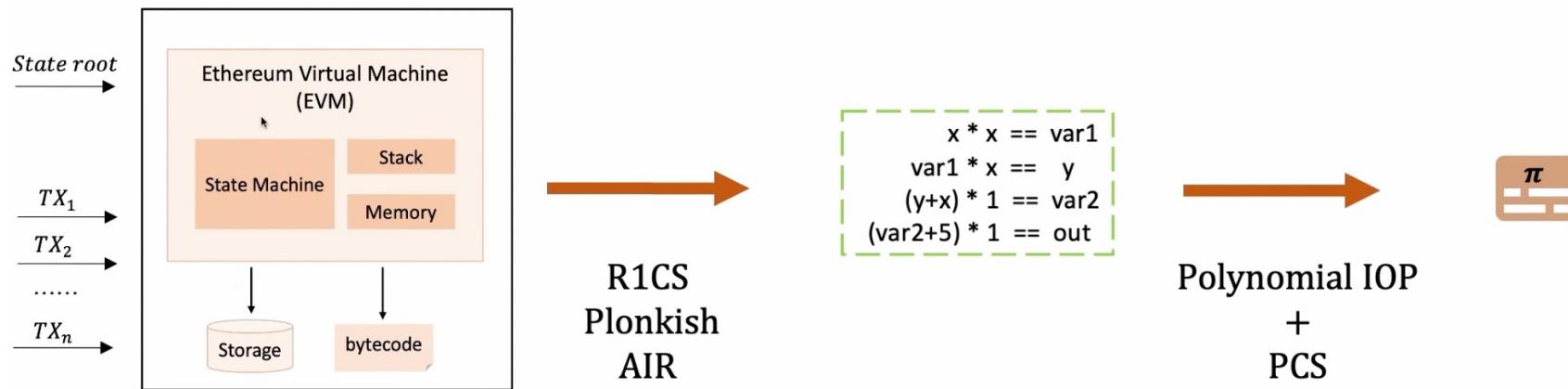


EVM presenta però delle sfide da affrontare:

1. La dimensione di una parola e ogni variabile in EVM è 256 bit, e questo è problematico per circuiti ZK che richiedono un campo finito dato una curva ellittica. Esmpi sono dei capi definiti su 254 bit. È pertanto necessario controllare che sia rispettato il range limit in maniera efficiente.
2. EVM ha molti opcode poco «amichevoli» come Keccak, sha-256, che richiedono circuiti molto ampi e complessi per la prova di questi opcodes. Pertanto, è necessario avere un modo efficiente di connettere circuiti.

::: ZK-Rollup in Blockchain (4/7)

L'impiego di una soluzione di prova implica la scrittura di un programma, la sua traduzione in uno dei possibili formati di rappresentazione e l'impiego di schemi di commitment e IOP per la generazione della prova.

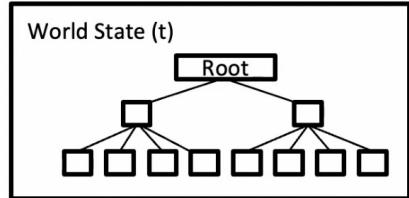


EVM presenta però delle sfide da affrontare:

3. Bisogna provare che quando viene letto qualcosa, il risultato ottenuto corrisponde al valori scritto in una precedente operazione.
4. EVM è caratterizzati da esecuzioni dinamiche e quindi sono necessari dei selettori on/off efficienti per abilitare diversi vincoli a diverse posizioni.

Per le prime 3 sfide si rendono necessari vincoli di lookup nell'aritmetica in uso per la definizione di vincoli, mentre per l'ultima sfida si rende necessario qualche gate custom che funga da selettore. Per questo si preferisce la sintassi Plonkish.

::: ZK-Rollup in Blockchain (5/7)

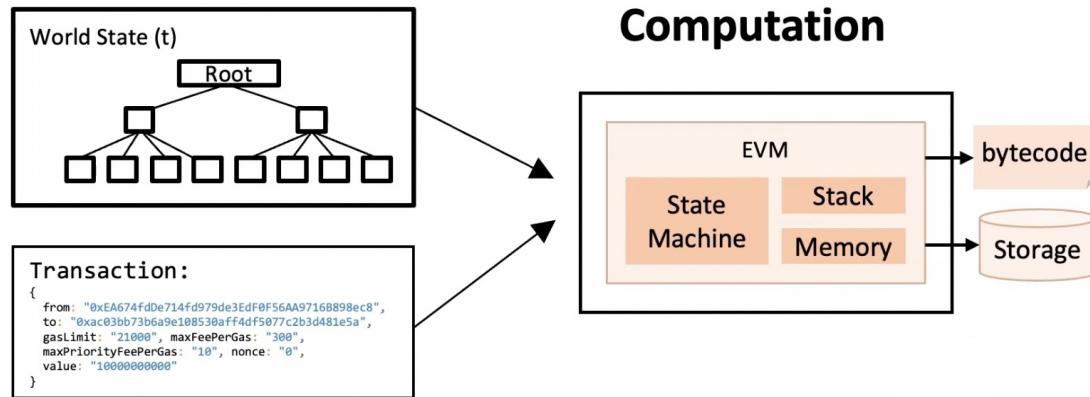


Transaction:

```
{  
  from: "0xE674fdDe714fd979de3EdF0F56AA9716B898ec8",  
  to: "0xac03b73b6a9e108530aff4df5077cb3d481e5a",  
  gasLimit: "21000", maxFeePerGas: "300",  
  maxPriorityFeePerGas: "10", nonce: "0",  
  value: "1000000000"  
}
```

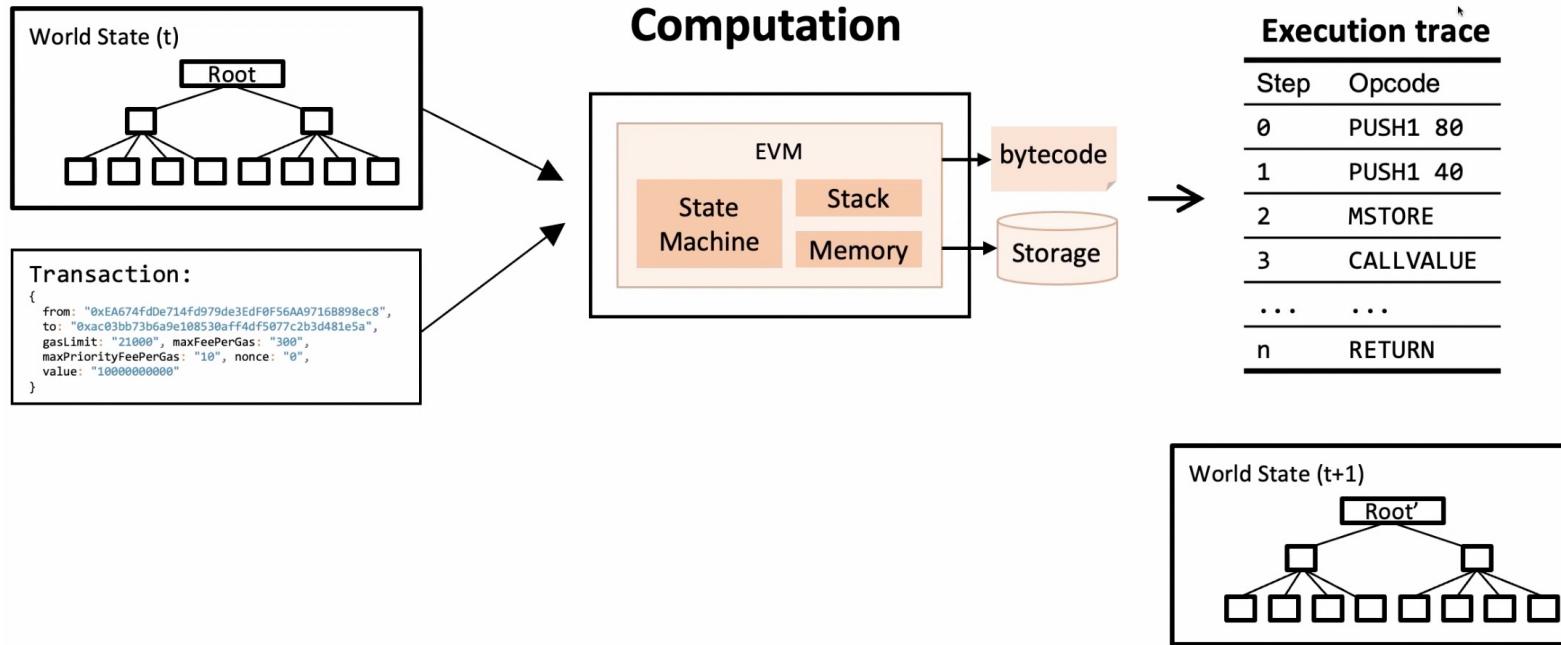
Inizialmente si ha lo stato iniziale della EVM e una transazione che si vuole eseguire.

::: ZK-Rollup in Blockchain (5/7)



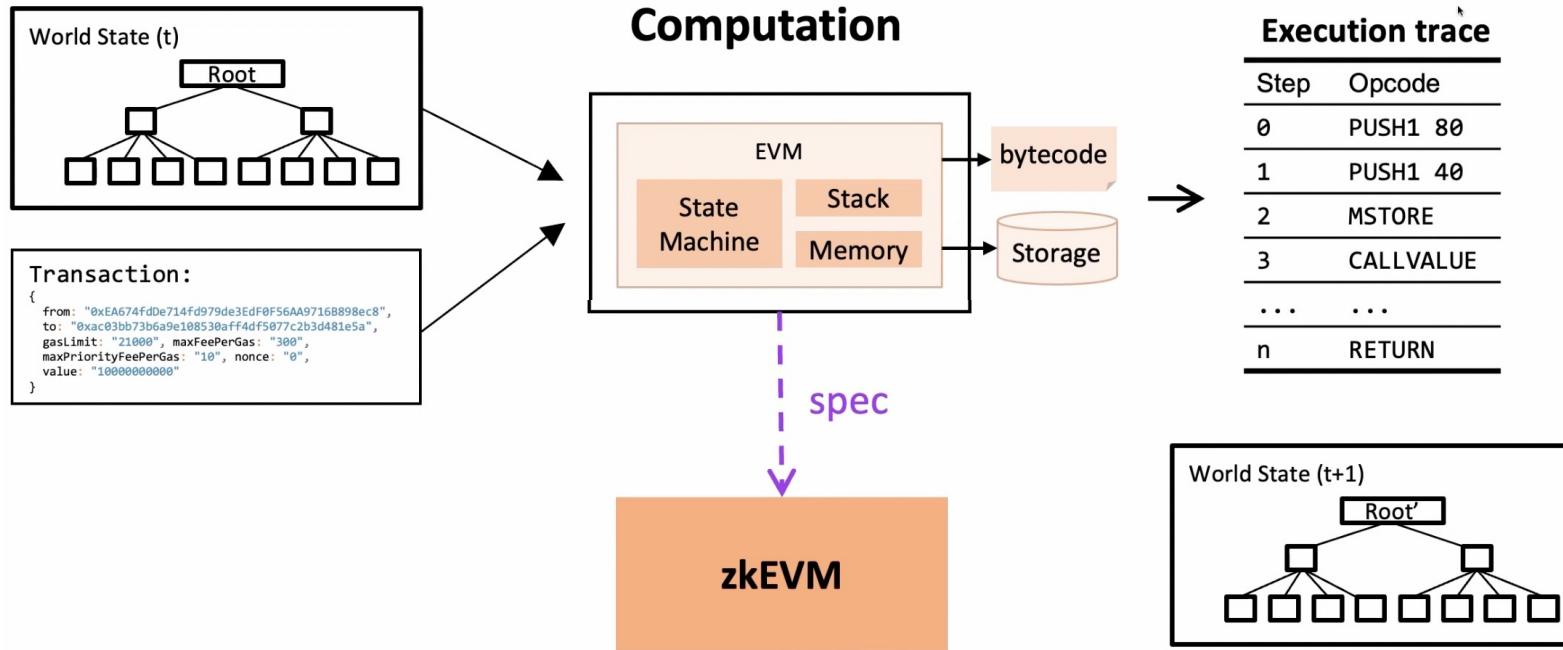
Alla ricezione della transazione, il nodo eseguirà la transazione sulla EVM, che caricherà il bytecode del contratto e lo eseguirà iterando sui vari opcodes...

... ZK-Rollup in Blockchain (5/7)



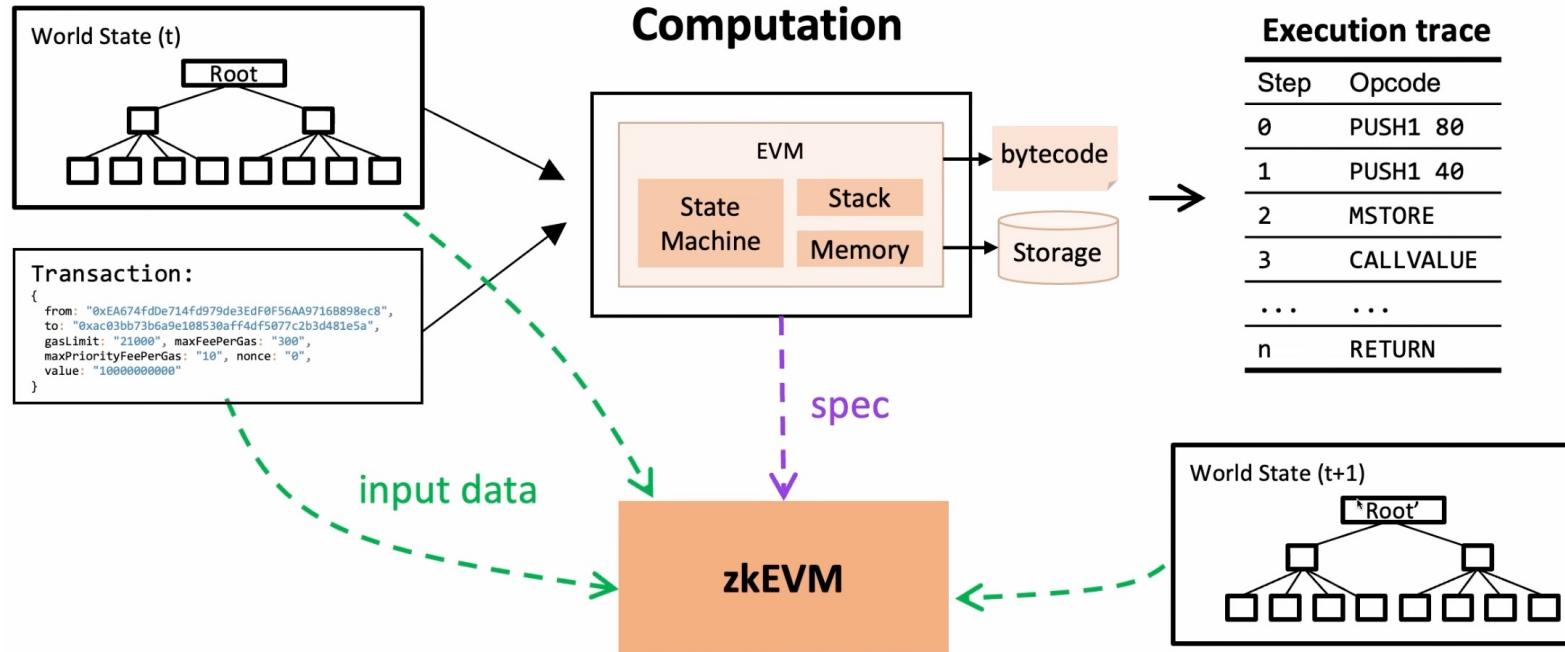
Alla ricezione della transazione, il nodo eseguirà la transazione sulla EVM, che caricherà il bytecode del contratto e lo eseguirà iterando sui vari opcodes ottenendo un execution trace (ovvero la sequenza degli opcode eseguiti) e una variazione del World state.

::: ZK-Rollup in Blockchain (5/7)



Bisogna definire i vincoli per tutte le operazioni che possono avvenire in EVM, per ogni possibile opcode, per formalizzare che sono valide.

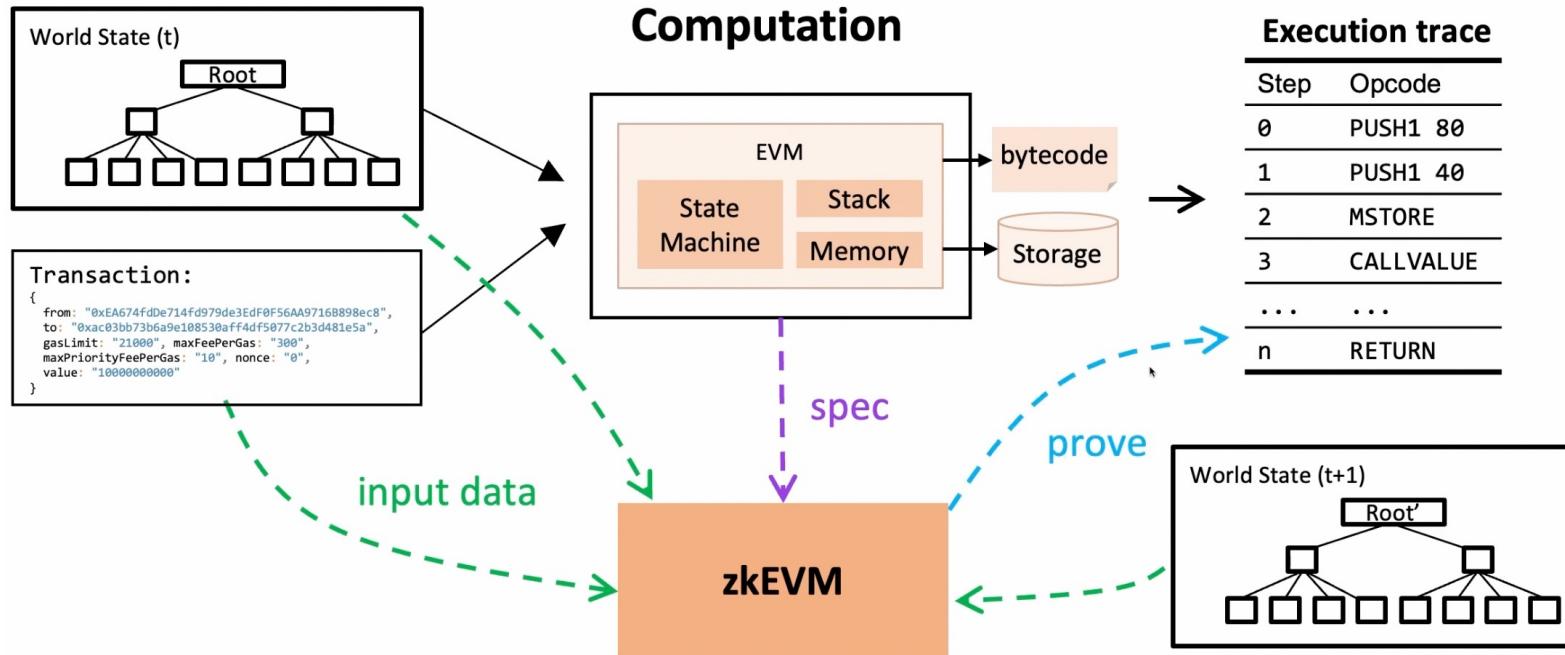
... ZK-Rollup in Blockchain (5/7)



L'input pubblico per zkEVM è rappresentato dallo stato precedente e successivo all'esecuzione della transazione, congiuntamente alla transazione (o alle N transazioni) che hanno portato all'evoluzione dall' stato t a quello t + 1.

Il verifier non deve conoscere concretamente l'esecution trace o cosa è stato eseguito durante dell'esecuzione della transazione (o delle N transazioni).

... ZK-Rollup in Blockchain (5/7)

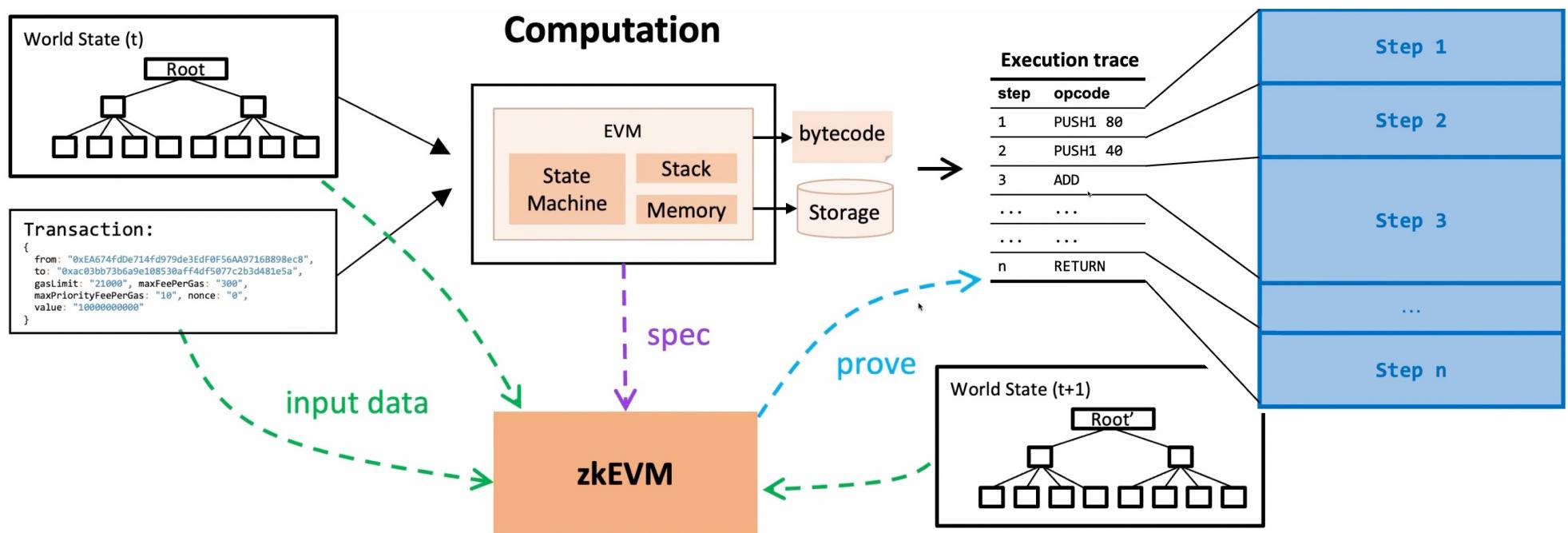


L'input pubblico per zkEVM è rappresentato dallo stato precedente e successivo all'esecuzione della transazione, congiuntamente alla transazione (o alle N transazioni) che hanno portato all'evoluzione dall' stato t a quello $t + 1$.

Il verifier non deve conoscere concretamente l'esecution trace o cosa è stato eseguito durante dell'esecuzione della transazione (o delle N transazioni).

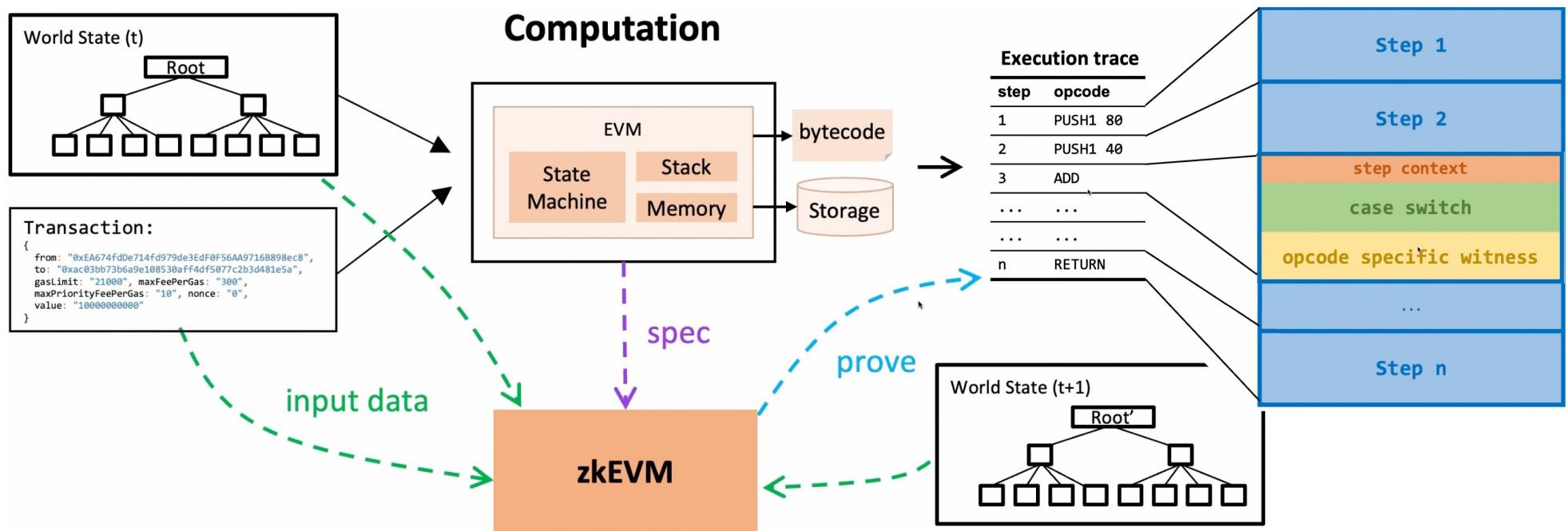
zkEVM prende l'execution trace come testimone privato, e convincere il verifier della corretta esecuzione, e quindi che lo stato a $t + 1$ è valido.

... ZK-Rollup in Blockchain (5/7)



Solo la lista di opcode nell'execution trace non basta ma bisogna estenderlo in una tabella di esecuzione più grande, che può avere forma di tabella nell'aritmetizzazione Plonkish ed aiutare nella definizione di vincoli.

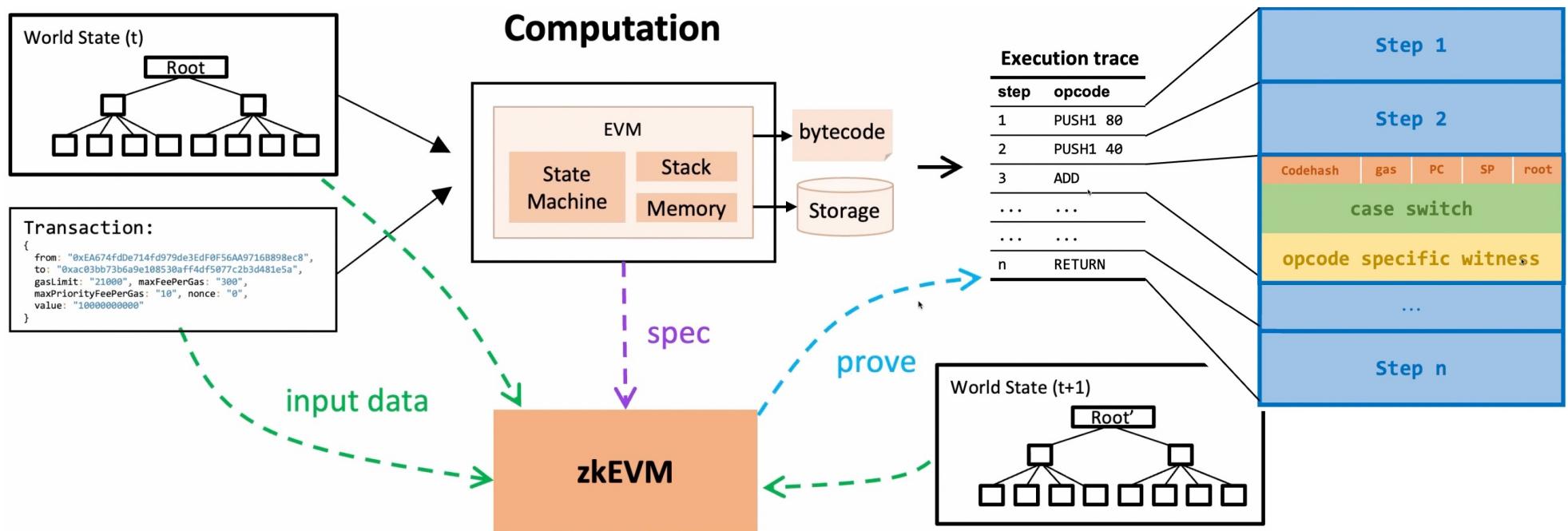
... ZK-Rollup in Blockchain (5/7)



Solo la lista di opcode nell'execution trace non basta ma bisogna estenderlo in una tabella di esecuzione più grande, che può avere forma di tabella nell'aritmetizzazione Plonkish ed aiutare nella definizione di vincoli.

In ogni step, ci sono tre tipologie di testimoni:

... ZK-Rollup in Blockchain (5/7)

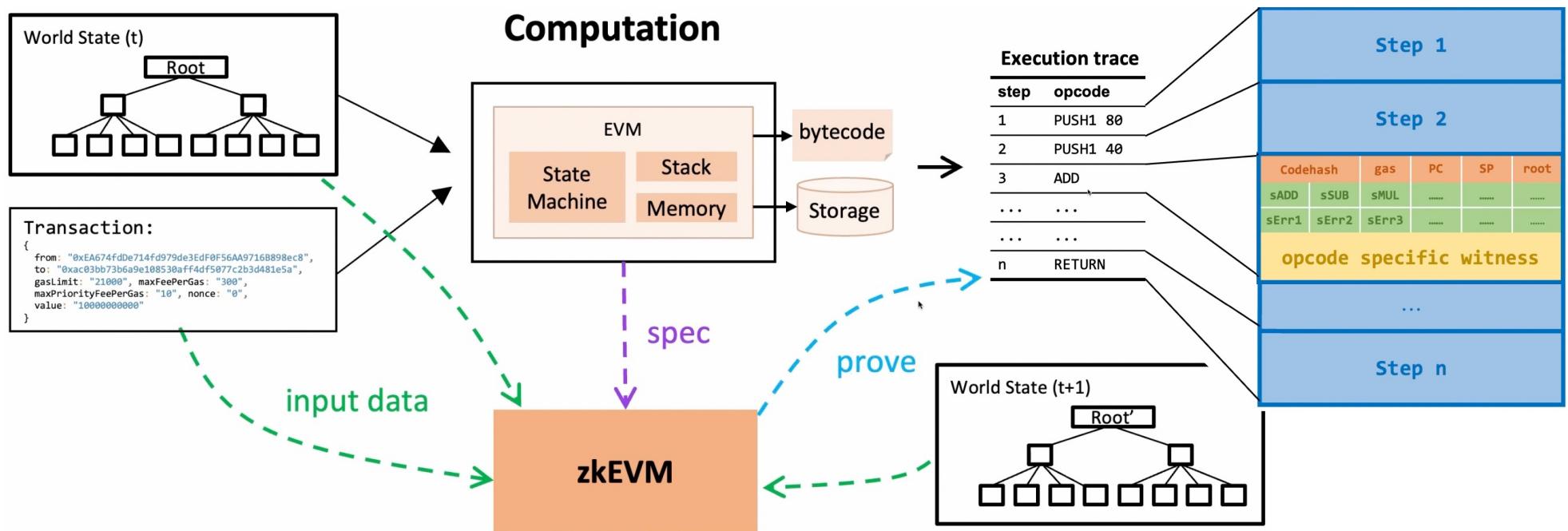


Solo la lista di opcode nell'execution trace non basta ma bisogna estenderlo in una tabella di esecuzione più grande, che può avere forma di tabella nell'aritmetizzazione Plonkish ed aiutare nella definizione di vincoli.

In ogni step, ci sono tre tipologie di testimoni:

- Lo step contest: quando si avanza nell'esecuzione, il contesto include l'hash del codice da eseguire, il gas rimanente dopo l'esecuzione, il program counter, lo stack pointer, il root del Merkle tree del World state, ovvero lo stato della VM all'esecuzione di questo passo di avanzamento.

... ZK-Rollup in Blockchain (5/7)

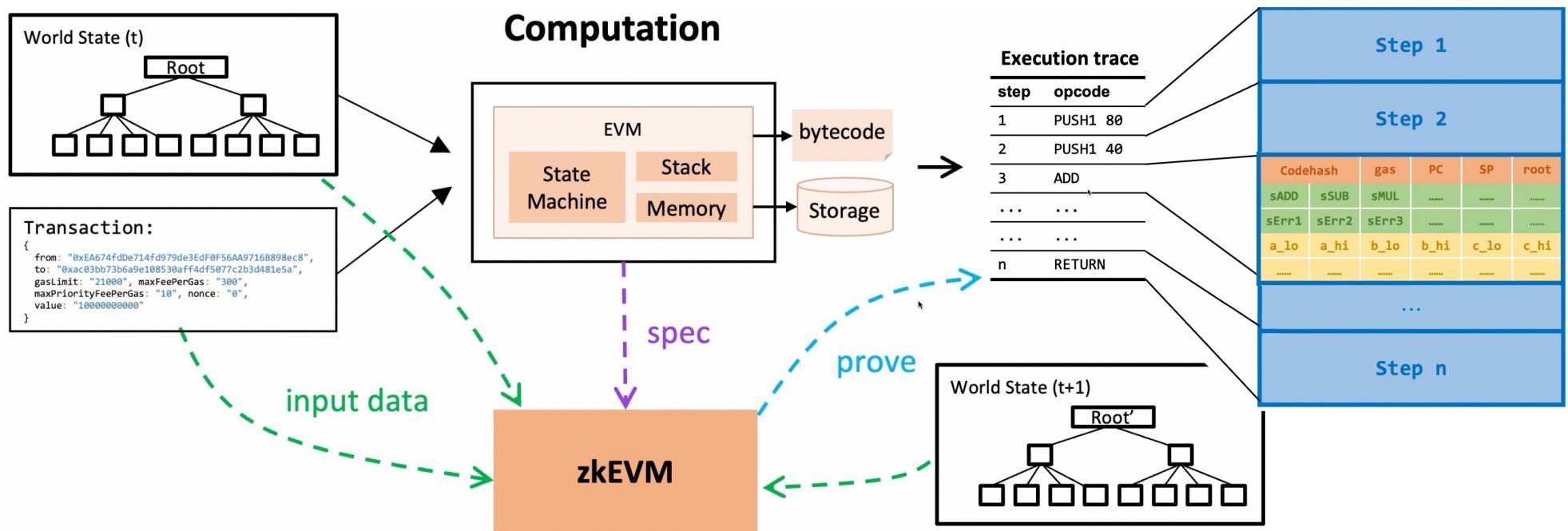


Solo la lista di opcode nell'execution trace non basta ma bisogna estenderlo in una tabella di esecuzione più grande, che può avere forma di tabella nell'aritmetizzazione Plonkish ed aiutare nella definizione di vincoli.

In ogni step, ci sono tre tipologie di testimoni:

- Il case switch: contiene tutte i possibili opcode e i casi di errore per uno step di avanzamento. Se si vuole eseguire la somma (Add), quindi la variabile associata all'opcode relativo deve essere 1 mentre le variabili per gli altri opcode devono essere 0. Rappresenta il selettore del vincolo da applicare a questo passo.

... ZK-Rollup in Blockchain (5/7)

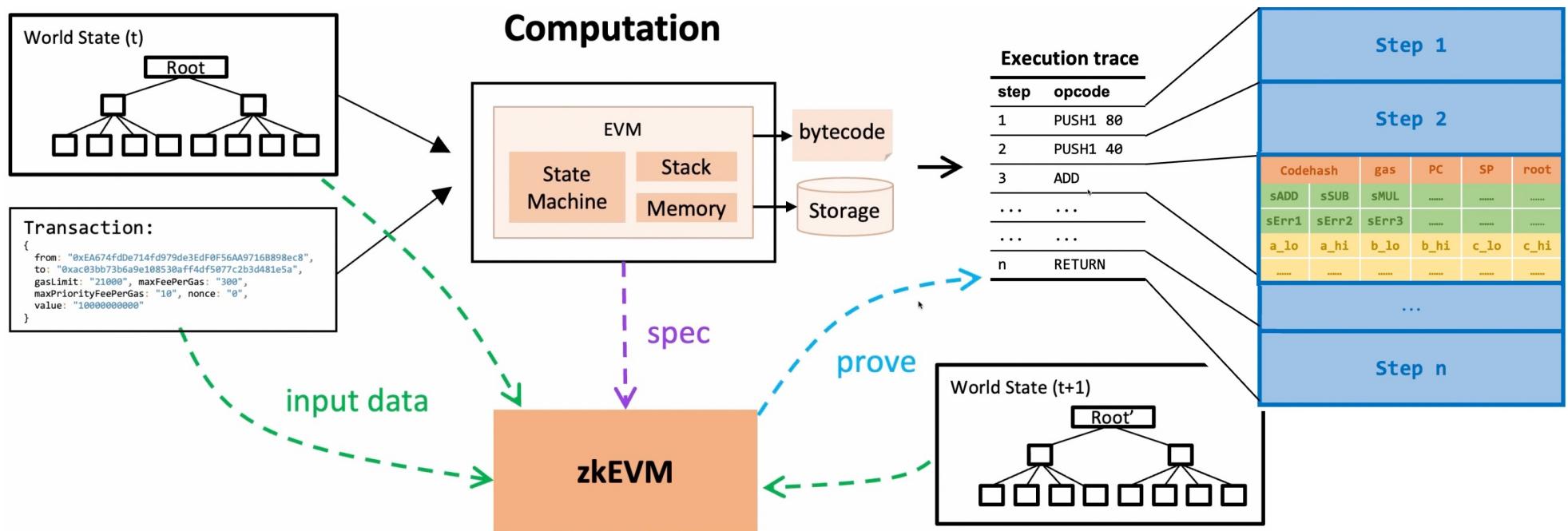


Solo la lista di opcode nell'execution trace non basta ma bisogna estenderlo in una tabella di esecuzione più grande, che può avere forma di tabella nell'aritmetizzazione Plonkish ed aiutare nella definizione di vincoli.

In ogni step, ci sono tre tipologie di testimoni:

- L'opcode specific witness: rappresenta tutto il testimone specifico per questo passo di avanzamento, ad es. nel caso della somma si necessitano gli operandi.

... ZK-Rollup in Blockchain (5/7)

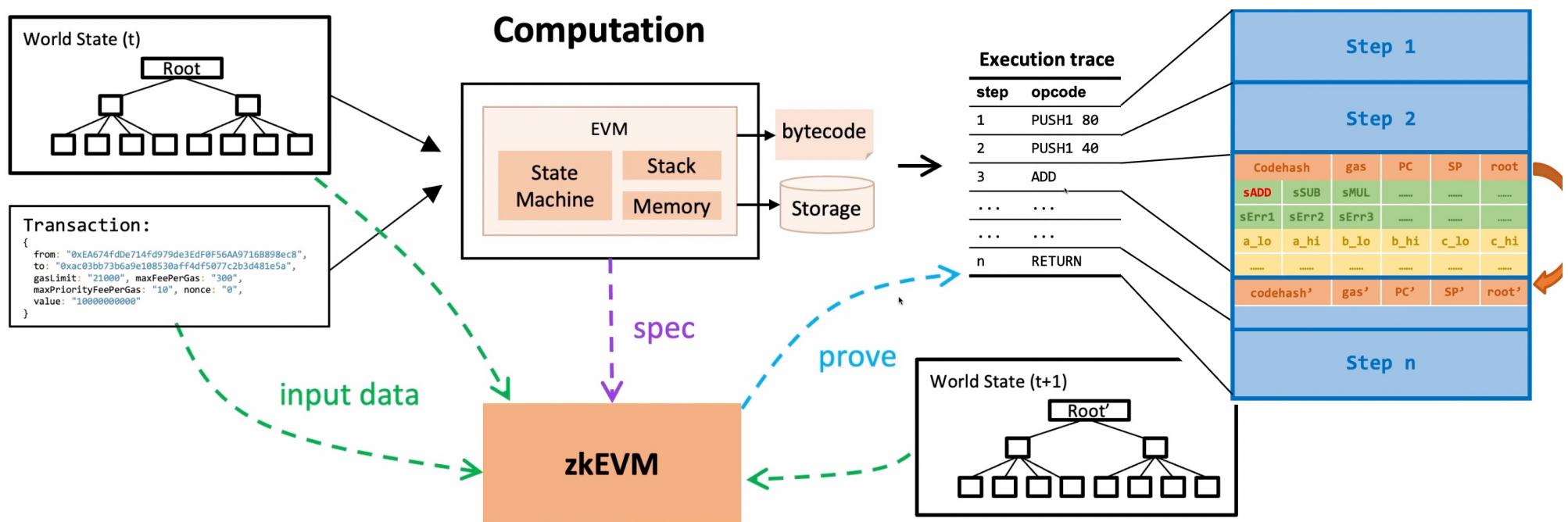


Solo la lista di opcode nell'execution trace non basta ma bisogna estenderlo in una tabella di esecuzione più grande, che può avere forma di tabella nell'aritmetizzazione Plonkish ed aiutare nella definizione di vincoli.

In ogni step, ci sono tre tipologie di testimoni:

- L'opcode specific witness: rappresenta tutto il testimone specifico per questo passo di avanzamento, ad es. nel caso della somma si necessitano gli operandi.

... ZK-Rollup in Blockchain (5/7)



Vediamo un esempio di vincoli con ADD.

- pc, sp e gas devono rispettare i vincoli di contesto tra uno step e l'altro.
- Inoltre solo la somma deve essere abilitata nei vincoli del case switch, e la variabile di scelta deve essere booleana
- L'ultimo vincolo specifica il comportamento della somma.

- **Step context**

$$\begin{aligned} sADD * (pc' - pc - 1) &= 0 \\ sADD * (sp' - sp - 1) &= 0 \\ sADD * (gas' - gas - 3) &= 0 \end{aligned}$$

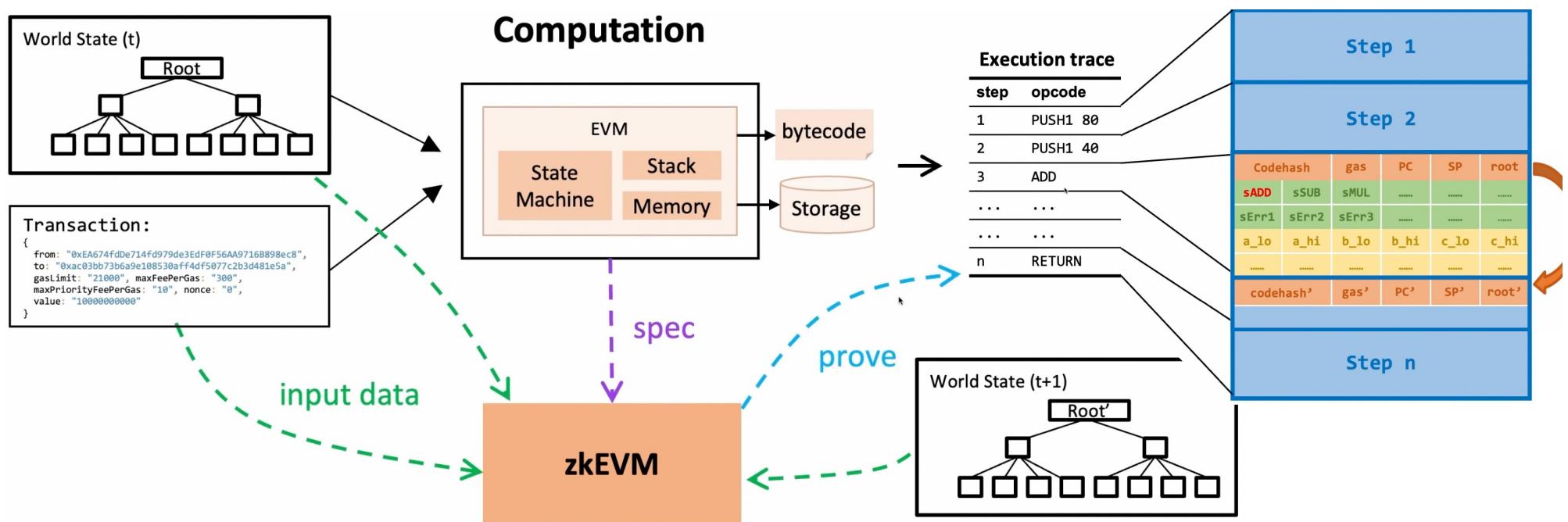
- **Case switch**

$$\begin{aligned} sADD * (1 - sADD) &= 0 \\ sMUL * (1 - sMUL) &= 0 \\ \dots \\ sADD + sMUL + \dots + sERRk &= 1 \end{aligned}$$

- **Opcode specific witness**

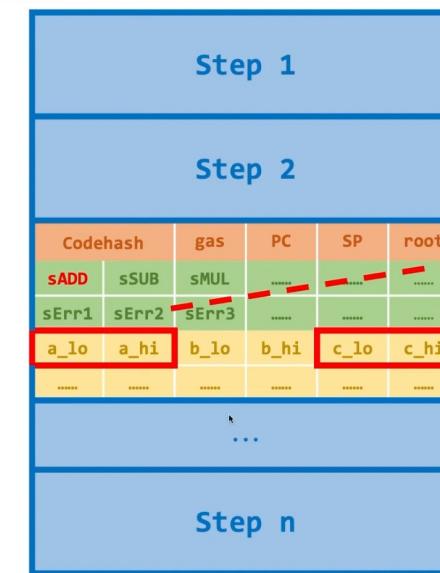
$$\begin{aligned} sADD * (a_lo + b_lo - c_lo - carry0 * 2^{128}) &= 0 \\ sADD * (a_hi + b_hi + carry0 - c_hi - carry1 * 2^{128}) &= 0 \end{aligned}$$

... ZK-Rollup in Blockchain (5/7)



Manca un'ultimo insieme di vincoli, ovvero ADD prende i suoi operandi dallo stack e passa il risultato sullo stack. Bisogna vincolare che effettivamente i valori siano stati presi dallo stack.

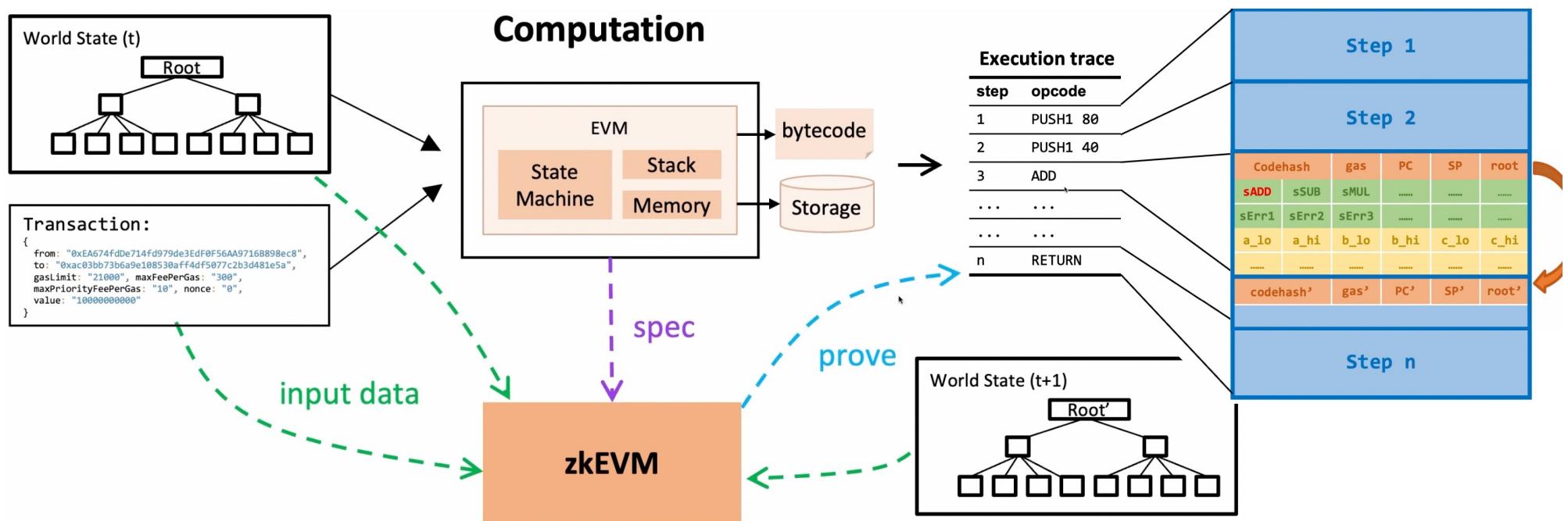
Questo si ha pensando di avere una tabella che contiene le scritture sequenziali sullo stack, e introducendo vincoli di lookup.



- **Opcode specific witness**

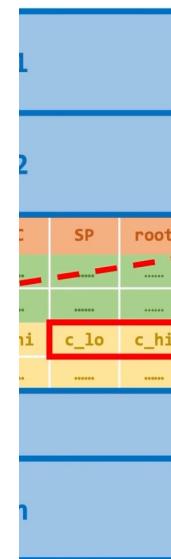
idx	tag	addr	R/W	value
1	STACK	1023	1	...
5	STACK	1022	0	word_a
6	STACK	1023	0	word_b
7	STACK	1023	1	word_c
...	STACK
...	MEMORY	0x40	1	...
...	MEMORY
...	STORAGE

... ZK-Rollup in Blockchain (5/7)



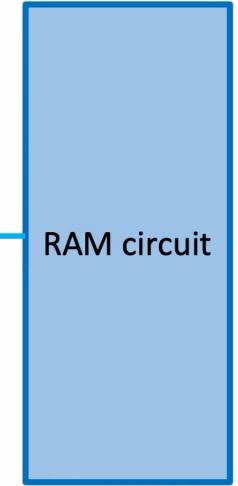
Manca un'ultimo insieme di vincoli, ovvero ADD prende i suoi operandi dallo stack e passa il risultato sullo stack. Bisogna vincolare che effettivamente i valori siano stati presi dallo stack.

La correttezza dei valori nella tabella sono provati da circuiti associati che validano la correttezza delle informazioni.

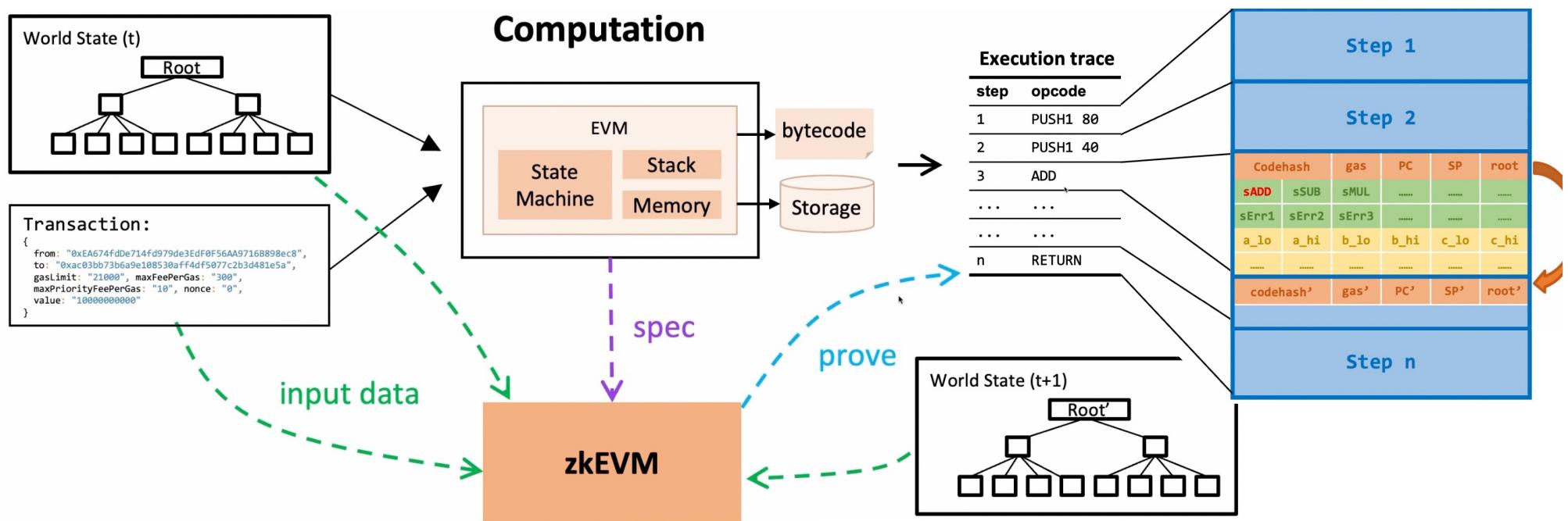


- **Opcode specific witness**

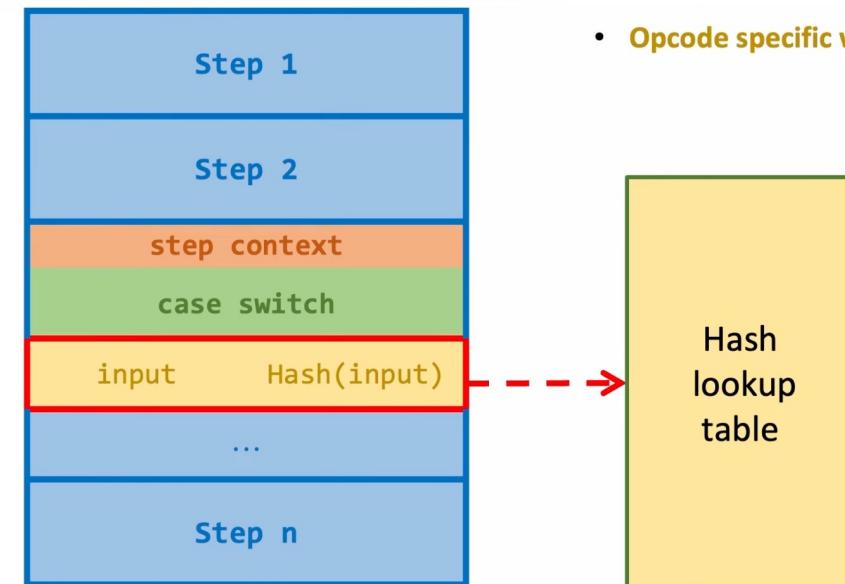
idx	tag	addr	R/W	value
1	STACK	1023	1	...
5	STACK	1022	0	word_a
6	STACK	1023	0	word_b
7	STACK	1023	1	word_c
...	STACK
...	MEMORY	0x40	1	...
...	MEMORY
...	STORAGE



... ZK-Rollup in Blockchain (5/7)

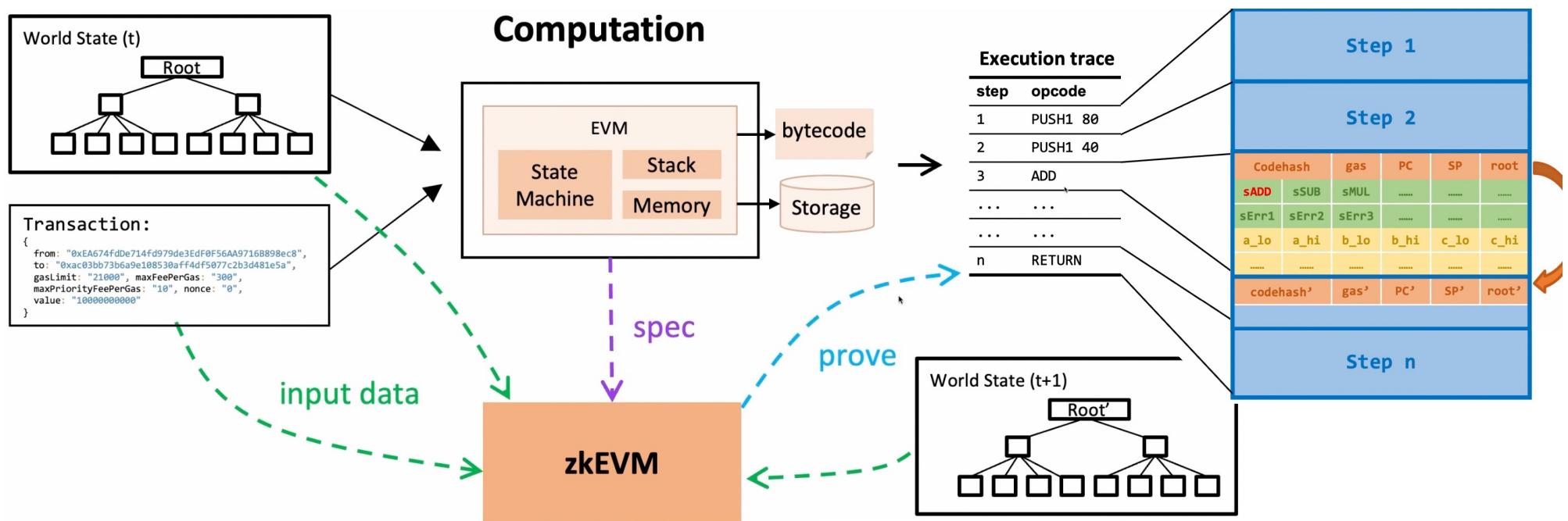


Nel caso di operazioni complesse come una funzione di Hash, si utilizza una tabella di lookup che antiene le coope di possibili input ed output, e se ad uno step si trova una coppia nella tabella, allora l'operazione è svolta correttamente.



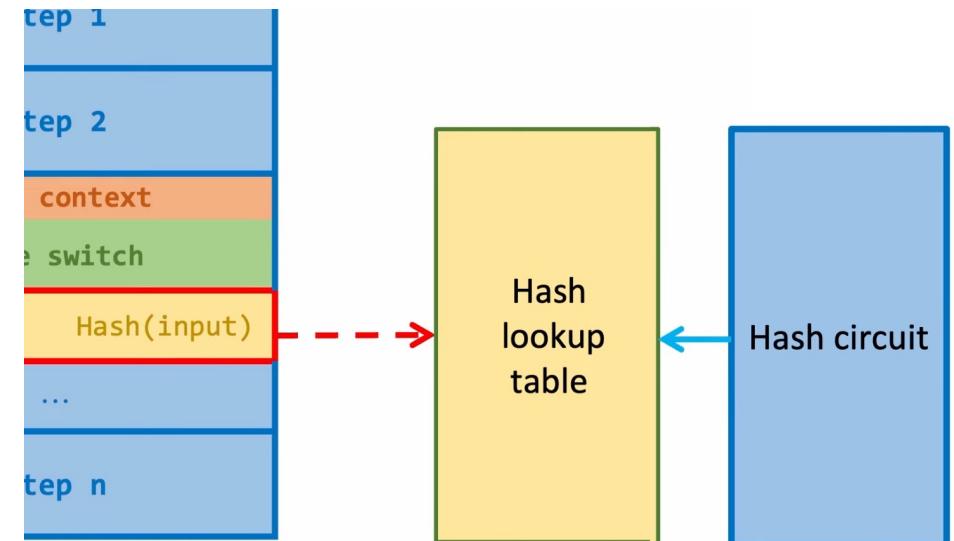
- Opcode specific witness

... ZK-Rollup in Blockchain (5/7)

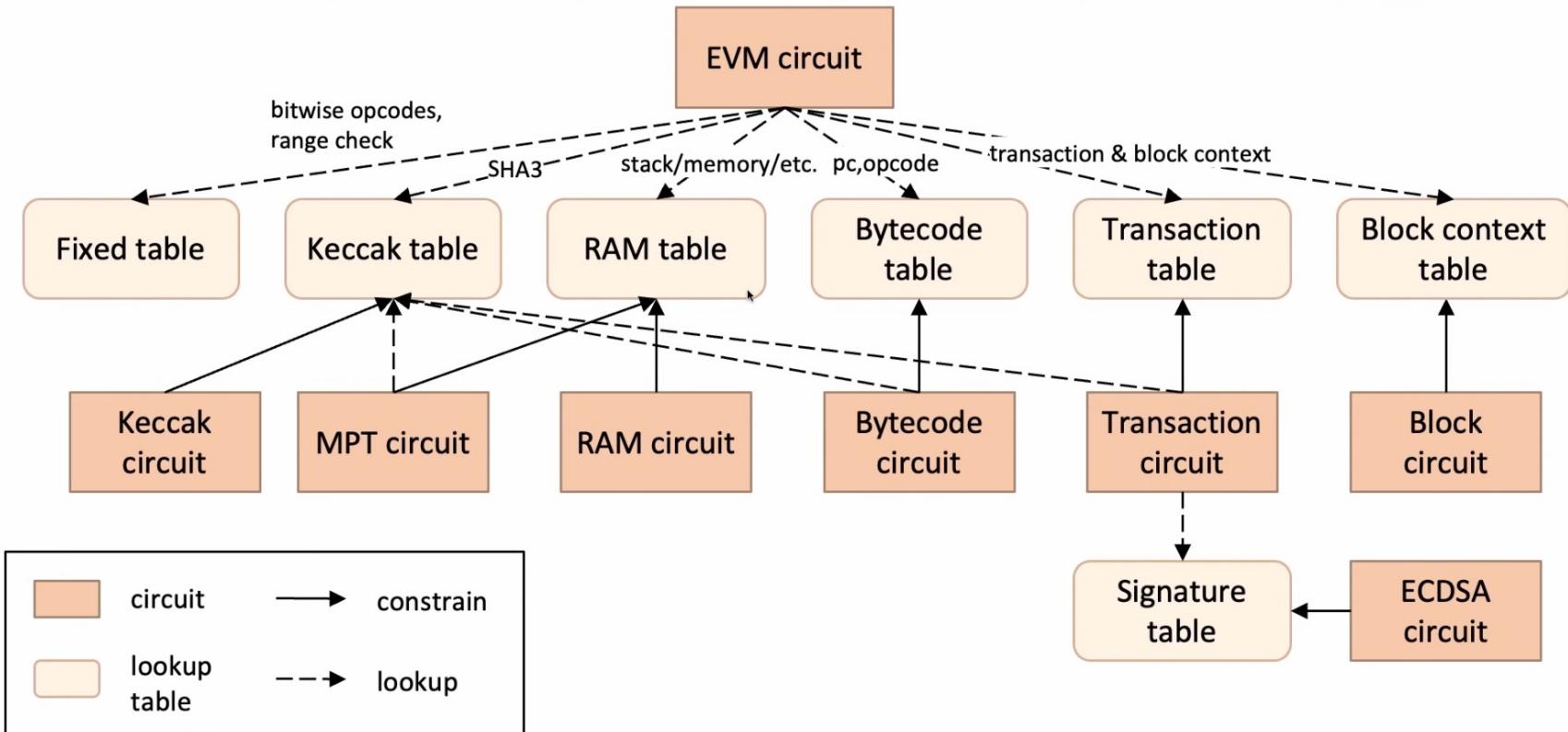


Nel caso di operazioni complesse come una funzione di Hash, si utilizza una tabella di lookup che antiene le coope di possibili input ed output, e se ad uno step si trova una coppia nella tabella, allora l'operazione è svolta correttamente.

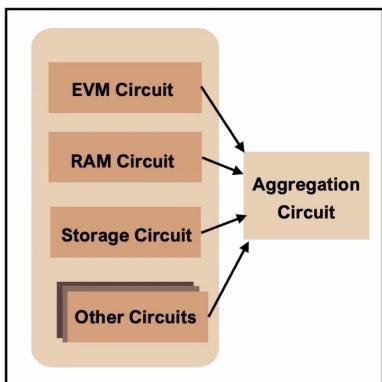
Anche in questo caso, appositi circuiti provano la correttezza delle associazioni.



::: ZK-Rollup in Blockchain (6/7)

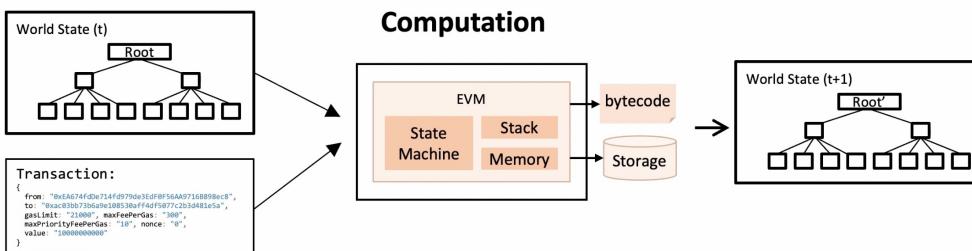
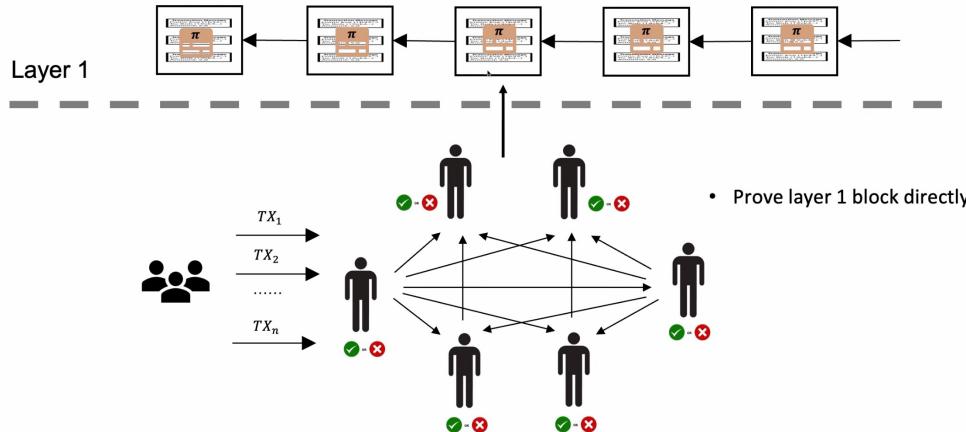


zkEVM



Una zkEVM si compone di vari circuiti e una serie di prove per tutti i circuiti disponibili. Inviare tutte queste prove sarebbe proibitivo, pertanto ci dispone di un circuito di aggregazione per aggregare tutte le prove, in un'unica che verifica la correttezza delle transazioni.

... ZK-Rollup in Blockchain (7/7)



È possibile usare prove ZK per migliorare il meccanismo di consenso evitando che ogni validatore debba rieseguire le transazioni.

È difficile perché richiede zkEVM di terzo tipo o completamente EVM compatibili, ed è ancora argomento di ricerca.

Si posso usare queste prove mantenendo la transazione nel testimone. E non nell'input pubblico. Questo è un modo per rendere le transazioni segrete e mettendo in blockchain solo un loro hash. Oppure è possibile implementare il proof-of-exploitation per dimostrare la conoscenza di un bug nello smart contract che con una data transazione può svuotare un balance.