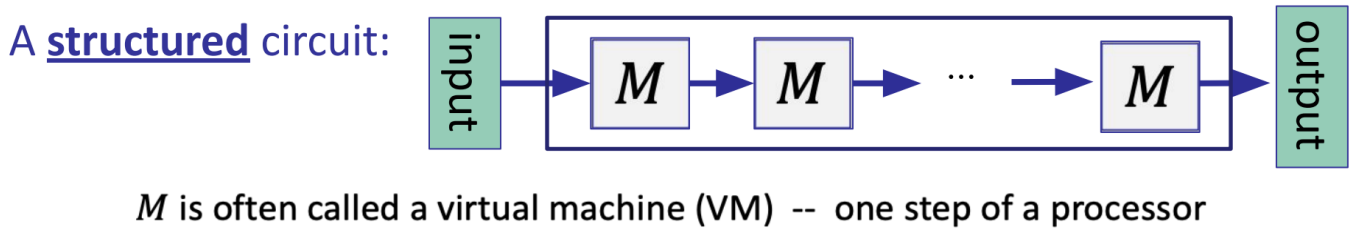
SNARK intro.

$|C| = \#\ gates\ in\ C\ (Circuit)$

$\xleftarrow{\$}$ means sample uniformly from the finite field.

Structured vs. unstructured circuits :

- An **unstructured** circuit: a circuit with arbitrary wires
- A **structured** circuit :



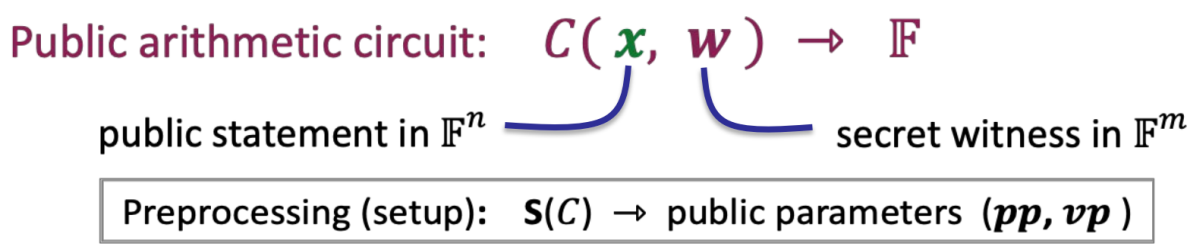
M is often called a virtual machine (VM) -- one step of a processor

structured circuit is **fixed**, and input just **repeated over and over again**. also called Virtual Machine .

> Some SNARK techniques only apply to structured circuits

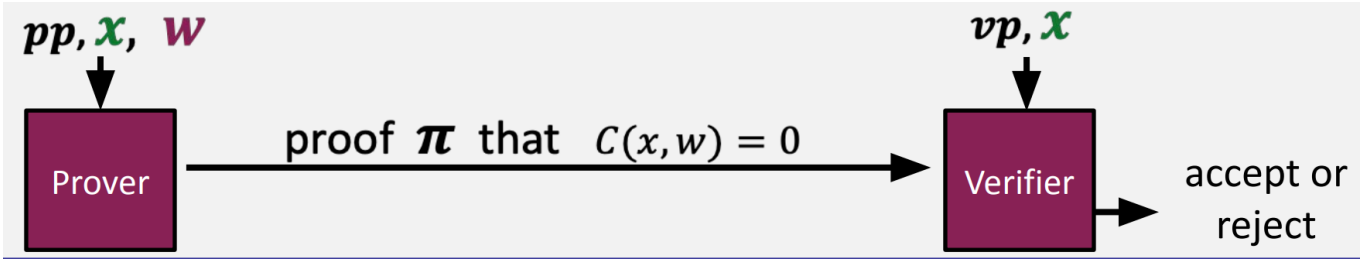
NARK:

NARK means Non-interactive ARgument of Knowledge.



- C : circuit
- x : public statement
- w : $secret$ *witness*
- **Preprocessing** is alse called **setup**, it takes a description of the circuit as input, then it outputs these public parameters, which we'll call pp & vp :
  - pp means **public** params for Prover.
  - vp means **public** params for Verifier.

NARK :



Prover and Verifier will each take their own inputs :

- prover takes the x (public statement) & pp (public (circuit)params) & the Witness
- Verifier takes vp & x (public statement)

Then the prover is trying to convince the verifier that **it knows some** w **such that** $C(x,\ w) = 0$

NARK Definition : A pre-processing NARK is a triple (S, P, V) , where :

- $S(C) \rightarrow$ generate the Circuit's $pp\ \&\ vp$ as public params for P & V.
- $P(pp, x, w) \rightarrow$: proof π

- $V(vp, x, \pi) \rightarrow$: $Accept$ or $Reject$.

> *All algs. and adversary have access to **random oracle** .*

SNARK:

SNARK: means a $Succinct$ ARgument of Knowledge

A succient preprocessing NARK is a triple (S, P, V) , where :

- $S(C) \rightarrow$ generate the Circuit's $pp \ \& \ vp$ as public params for P & V.
- $P(pp, x, w) \rightarrow$: short proof π ;
  - which means $len(\pi) = sublinear(|w|)$
  - We require that the length of the **proof** is sublinear in the length of w.
- $V(vp, x, \pi) \rightarrow$: fast to verify ;
  - $time(V) = O_\lambda(|x|, \ sublinear(|C|))$
  - ① the running time of the verifier should be sublinear in the size of the circuit. So the verifier **cannot simply rerun the circuit C**
  - ② It has to be linear in the statement x because the verifier has to at least read the statement x in order to know what it's verifying. So we allow it to be linear in x. haha.

> *example **sublinear**: like $f(n) = \sqrt{n}$.*

in reality we want to be more greedy, so a SNARK in practice actually is going to be $strongly \ \ succinct$.

A strongly succient preprocessing NARK is a triple (S, P, V) , where :

- $S(C) \rightarrow$ generate the Circuit's $pp \ \& \ vp$ as public params for P & V.
- $P(pp, x, w) \rightarrow$: shorter proof π ;
  - which means $len(\pi) = O_\lambda(\ \mathbf{log}|\mathbf{C}|\ )$
- $V(vp, x, \pi) \rightarrow$: fast to verify ;
  - $time(V) = O_\lambda(|x|, \ \mathbf{log}|\mathbf{C}|\ )$

$len(\pi) = O_\lambda(\ \mathbf{log}|\mathbf{C}|\ )$:

1. the time to verify should be at most **logarithmic** in the size of the circuit.
2. So we get very, very short proof even if we have very, very large circuits.

$time(V) = O_\lambda(|x|, \ \mathbf{log}|\mathbf{C}|\ )$:

1. you can realize the verifier actually doesn't even have time to read the circuit C !!!! it has to operate in time that's logarithmic and the size of the circuit C. (Verifier 甚至没有时间来读取整个电路)
2. So the verifier doesn't even know what the circuit C is. So how can it possibly verify a statement if it doesn't know what the underlying circuit is? (Verifier 不知道电路是什么样的, 又该如何去验证呢 ? 答: preprocessing)
3. And this is exactly why we need the preprocessing step. preprocessing is in some sense it reads the entire circuit C and **it generates a summary of the circuit C for the verifier**.

---

zk-SNARK :

- `SNARK` : a NARC (complete and knowledge sound) that is succinct
- `zk-SNARK` : a SNARK that is also **zero knowledge**

intuitively meaning that the SNARK proof reveals **nothing** about the witness.

The `trivial SNARK` is not a SNARK :

(a) Prover sends w to verifier,
(b) Verifier checks if $C(x, w) = 0$ and accepts if so.

Problems with this:

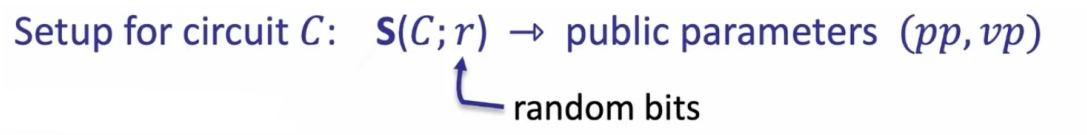
(1) w might be long: we want a "short" proof

(2) computing $C(x, w)$ may be hard: we want a "fast" verifier

(3) w might be secret: prover might not want to reveal w to verifier

Types of preprocessing Setup

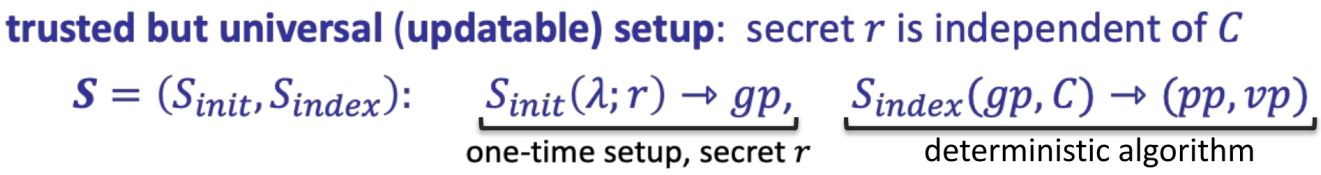
The preprocessing step often will **take some random bits**, they will use to generate parameters (pp; pv)



There are 3 types of **setup** :

① **trusted setup per circuit** :
1. the `setup` procedure has to be run afresh for **every circuit** that we want to preprocess.
2. And more importantly, it's really critical that the random bits r are kept secret from the prover. In particular, if the prover ever learns these random bits r , that it will allow it to **prove false statements**
3. that allows the prover to actually create money or steal, and so on.

② **trusted but universal (updatable)** setup :



1. split the set of procedure into 2 different procedures —— `Sinit` and `Sindex` .
2. `Sinit` is something that's run once and for all. generates global parameter gp
    1. gp 生成后, r 被销毁

③ **transparent setup** : $S(C)$ doesn't use secret data (no trusted setup)

Significant progress in recent years :

	size of proof π	verifier time	setup	post-quantum?
Groth'16	$\approx$ 200 Bytes $O_\lambda(1)$	$\approx$ 1.5 ms $O_\lambda(1)$	trusted per circuit	no
Plonk / Marlin	$\approx$ 400 Bytes $O_\lambda(1)$	$\approx$ 3 ms $O_\lambda(1)$	universal trusted setup	no
Bulletproofs	$\approx$ 1.5 KB $O_\lambda(\log \lvert C \rvert)$	$\approx$ 3 sec $O_\lambda(\lvert C \rvert)$	transparent	no
STARK	$\approx$ 100 KB $O_\lambda(\log^2 \lvert C \rvert)$	$\approx$ 10 ms $O_\lambda(\log^2 \lvert C \rvert)$	transparent	yes

> All of these Prover Time is almost linear in $|C|$

Definitions: knowledge soundness

> 没咋看懂这部分 ??

Formally: (S, P, V) is (adaptively) **knowledge sound** for a circuit C if

for every poly. time adversary $A = (A_0, A_1)$ such that

$gp \leftarrow S_{\text{init}}(\,),\quad (C, x, \text{st}) \leftarrow A_0(gp),\quad (pp, vp) \leftarrow S_{\text{index}}(C),\quad \pi \leftarrow A_1(pp, x, \text{st})$:

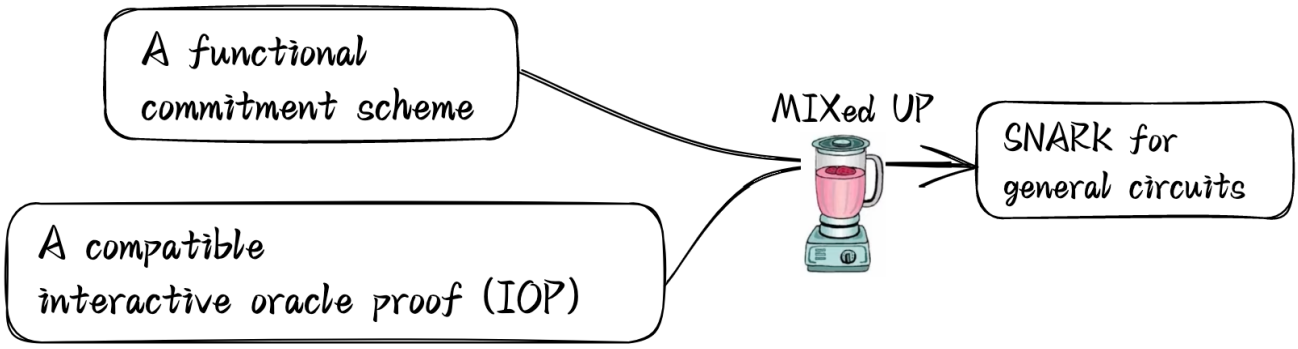
$\Pr[\, V(vp, x, \pi) = \text{accept}\,] > 1/10^6$ (non-negligible)

there is an efficient **extractor** E (that uses A) s.t.

$gp \leftarrow S_{\text{init}}(\,),\quad (C, x, \text{st}) \leftarrow A_0(gp),\quad w \leftarrow E(gp, C, x)$:

$\Pr[\, C(x, w) = 0\,] > 1/10^6 - \epsilon$ (for a negligible ϵ)

General paradigm (通用范例)



- **the functional commitment scheme** : is a cryptographic object, meaning that its security depends on certain cryptographic assumptions. (是一个密码学函数, 其安全性取决于密码学假设)
- **the interactive oracle proof** : IOP actually is an information theoretic object (信息论对象), so that we can prove security of an IOP unconditionally without any underlying assumption. (因此我们可以在没有任何底层假设的情况下无条件地证明 IOP 的安全性)

Later we will explain what each concept is.

Commitment

So a **commitment scheme** is made up of 2 algorithms : `commit` and `verify`

- $Commit(m,\ r) \rightarrow \mathbf{com}$ (r chosen at random)
- $Verify(m,\ \mathbf{com},\ r) \rightarrow Accept$ *or* $Reject$.

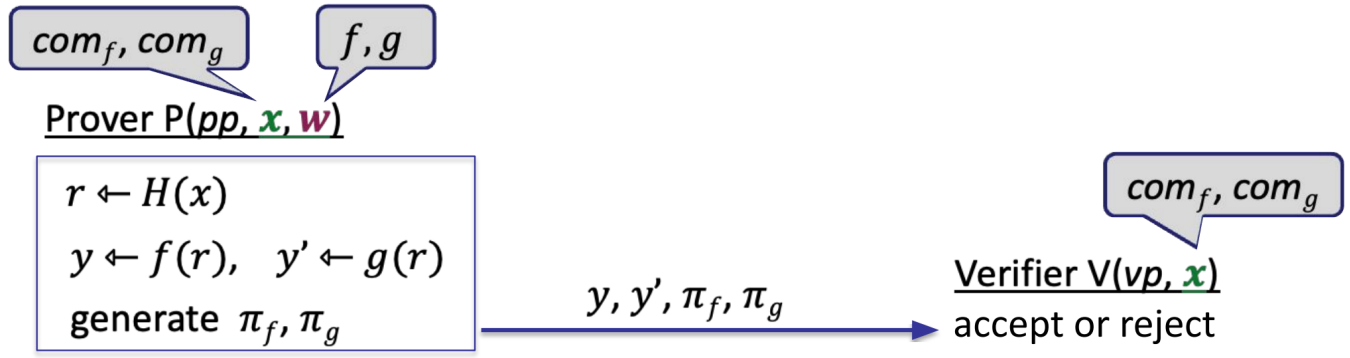
1. At a later time, the **committer** can open up the commitments by revealing the message m and r
2. And the verifier will run a verification algorithm that outputs either accept or reject.

Commitment schemes need to **satisfy 2 properties** :

1. **binding** : cannot produce $\mathbf{com}$ and two valid openings for $\mathbf{com}$
    1. it means once you've committed, you're bound to the message you committed.
2. **hiding** : $\mathbf{com}$ reveals nothing about commited data.
    1. so $\mathbf{com}$ reveals nothing about the message m.

Polynomial Commitment

Schwartz–Zippel lemma 略.

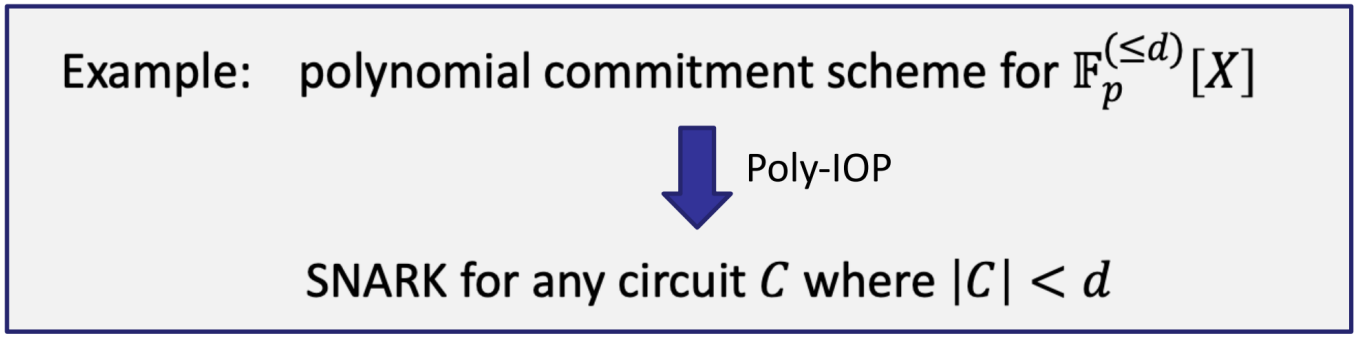


使用 Fiat-Shamir transform 将协议转化为**非交互式**：$Hash(com_f,\ com_g)$

Attention : this is just a SNARK, but *not* a **zk**-SNARK, because the verifier learns the value of the polynomials f and g at the point r . So the verifier learns something that it didn't know before.

IOP (Interactive Oracle Proof)

Goal: boost functional commitment ⇒ SNARK for general circuits
(functional commitment 是一种工具组件, IOP 就是在组件之上构建通用 SNARK Proof.)



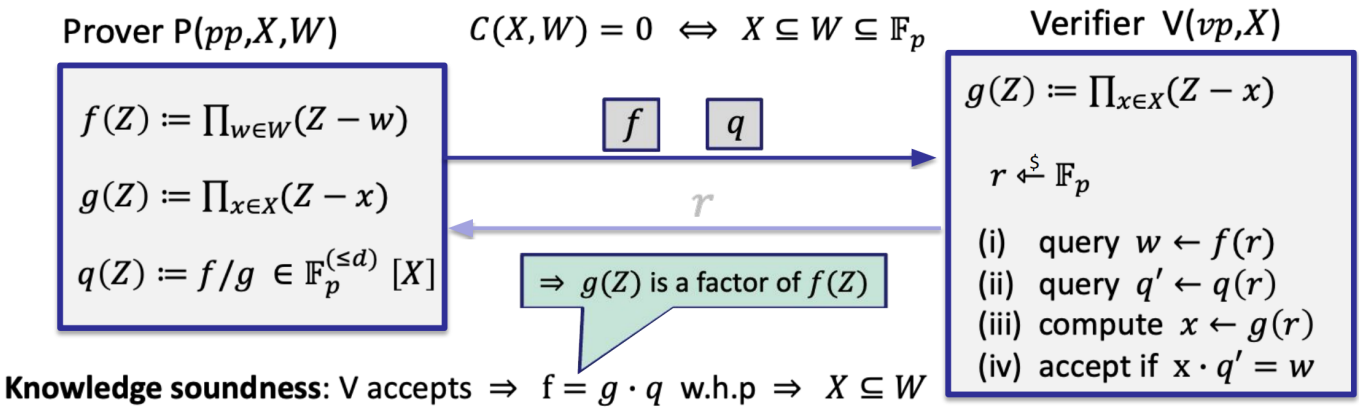
Example: polynomial commitment scheme for $\mathbb{F}_p^{\leq d}[X]$ (对于 $\mathbb{F}_p$ 上的低次多项式 (degree $< d$), 构建 polynomial commitment scheme)

经过 Poly-IOP 操作, 可以做到 :

SNARK for any circuit C where $|C| < d$ (for arbitrary circuit C ,如果 C 的规模小于 d , 都可以构建 SNARK 证明)

An Example Poly-IOP

This will be an example of an IOP(交互式证明), that's built on top of a polynomial commitment scheme. It's a bit contrived (人为构造, 略显矫揉造作)



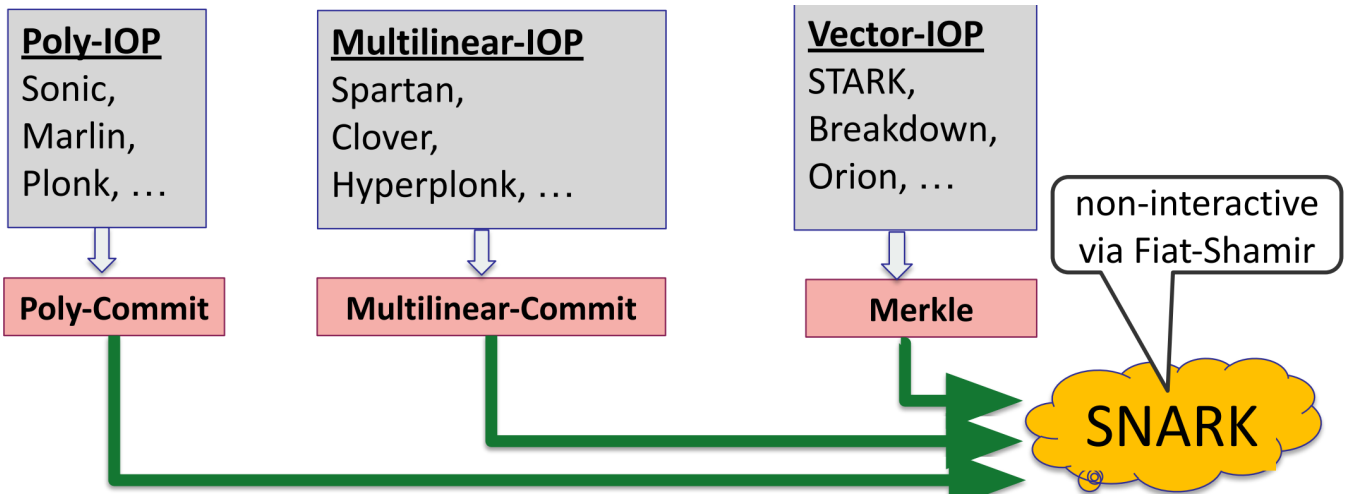
Knowledge soundness: V accepts ⇒ $f = g \cdot q$ w.h.p ⇒ $X \subseteq W$

Extractor(X,f,q,r): output witness W by computing all roots of $f(Z)$

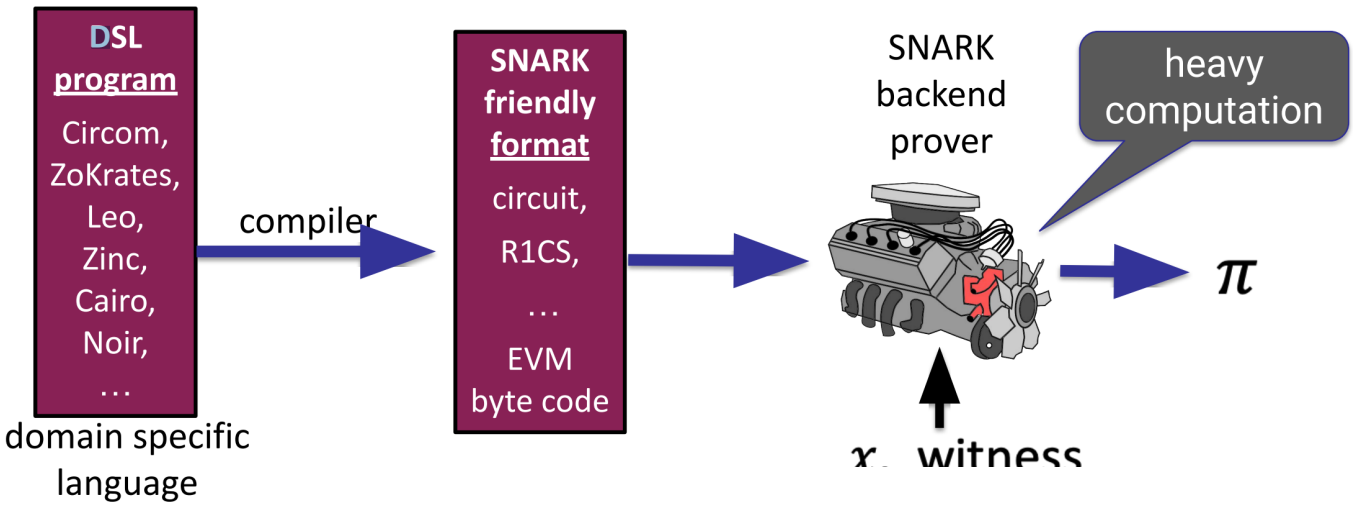
Prover wants to prove: $X \in W \in \mathbb{F}_p$, it can be represented as $C(X,\ W) = 0$
Prover 利用集合 W, X 构造 2 个多项式 $f(Z)$ 和 $g(Z)$

r 是虚线 : 这里的 r 后期可通过 Fiat-shamir 转化为非交互式

IOP Protocols



SNARKs in practice



---

1. You will likely do your programming projects in **Circom** (domain specific languages, DSL) to construct SNARK circuits.
2. And what this **compiler** does is it compiles the given program into a SNARK friendly format like a circuit / R1CS / EVM byte code .. and **that will be the actual input** to the SNARK backend system.
3. Backend Prover: where takes public statement x and the witness w and produces the **proof** π as output.

IOP(交互式证明系统)和Polynomial Commitment(多项式承诺)之间的关系:

1. Polynomial Commitment 是一种特殊的承诺方案,用于承诺一个低次数多项式。
2. IOP允许证明人与验证者进行交互,从而使验证者能够验证证明人确实知道某个隐藏值或者满足某个语句。
3. IOP通常依赖于某种承诺方案。例如在上面的例子中,IOP利用了Polynomial Commitment这种特殊的承诺方案。
4. 通过这种承诺方案,证明人可以承诺一个低次数的多项式,而不透露多项式的具体表达式。
5. 在IOP中,验证者可以通过交互,检查证明人是否真正知道这个隐藏的多项式。
6. 如果证明人能够满足验证者在交互中的所有查询,就证明了证明人确实知道这个多项式。
7. 将Polynomial Commitment与IOP结合,就可以实现对低次数多项式相关语句的验证。
8. 通过逐步增强Polynomial Commitment方案,可以扩展IOP所能验证的内容,最终实现对更一般电路和语句的证明。

总结来说,Polynomial Commitment为IOP提供了验证隐藏多项式表达式的手段。两者结合可以推广构建更一般的交互式证明系统。