APPLICATIONS OF INDISTINGUISHABILITY OBFUSCATION

Mark Zhandry – Stanford University

*Joint work with Dan Boneh

Program Obfuscation

Intuition: Scramble a program

- Same functionality as original
- Hides all implementation details

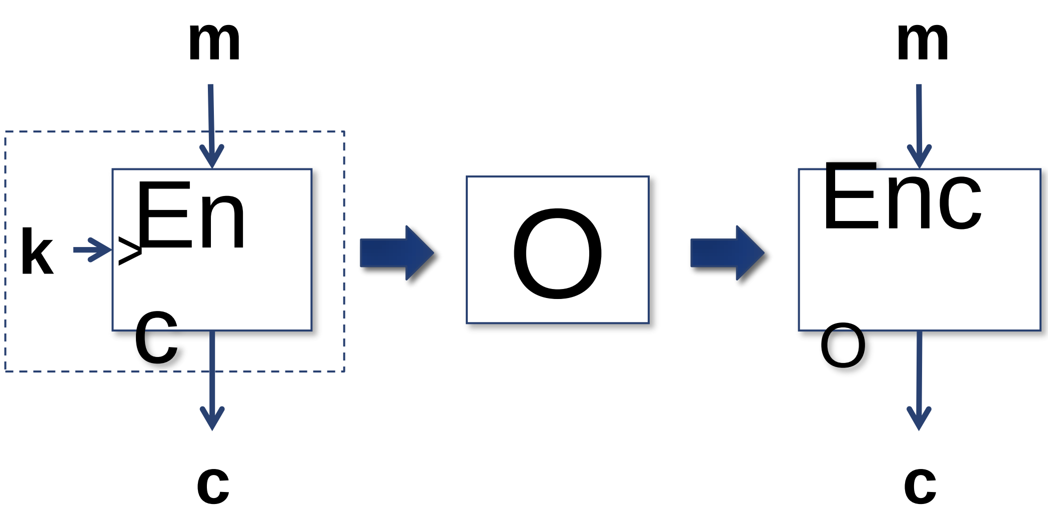
Potential uses:

- IP protection
- Prevent tampering
- **Cryptography**

Applications

Crypto

- Give out program with embedded secrets
- Obfuscate to hide secrets
- Ex: symmetric key to public key encryption



Keygen:

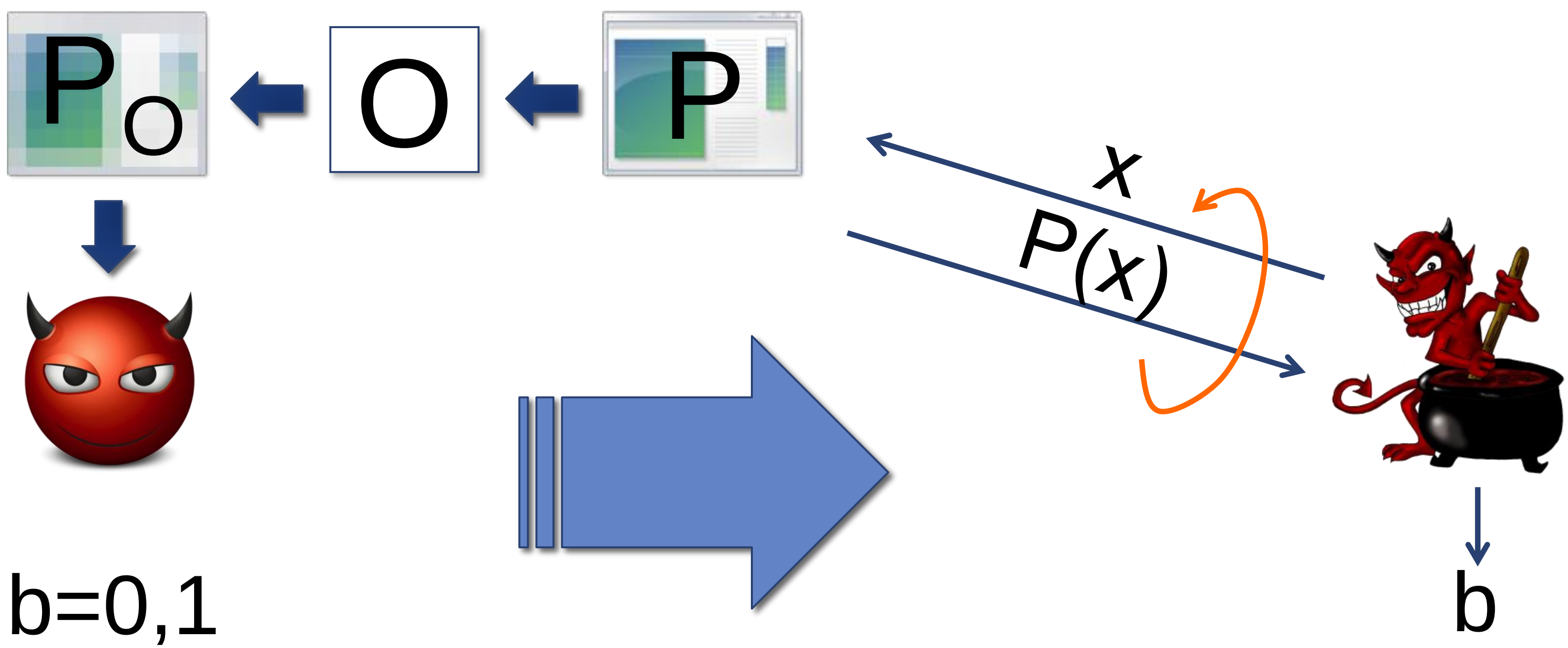
- Publish **Enc**$_{\mathbf{O}}$ as **pk**
  ⇒ **k** remains secret
- Keep **k** as **sk**

Virtual Black Box (VBB) Obfuscation [BGIRSVY'01]

What can we learn about **P** from an obfuscation $\mathbf{P_O}$?

- Output on any input
- Anything derivable from polynomial number of outputs

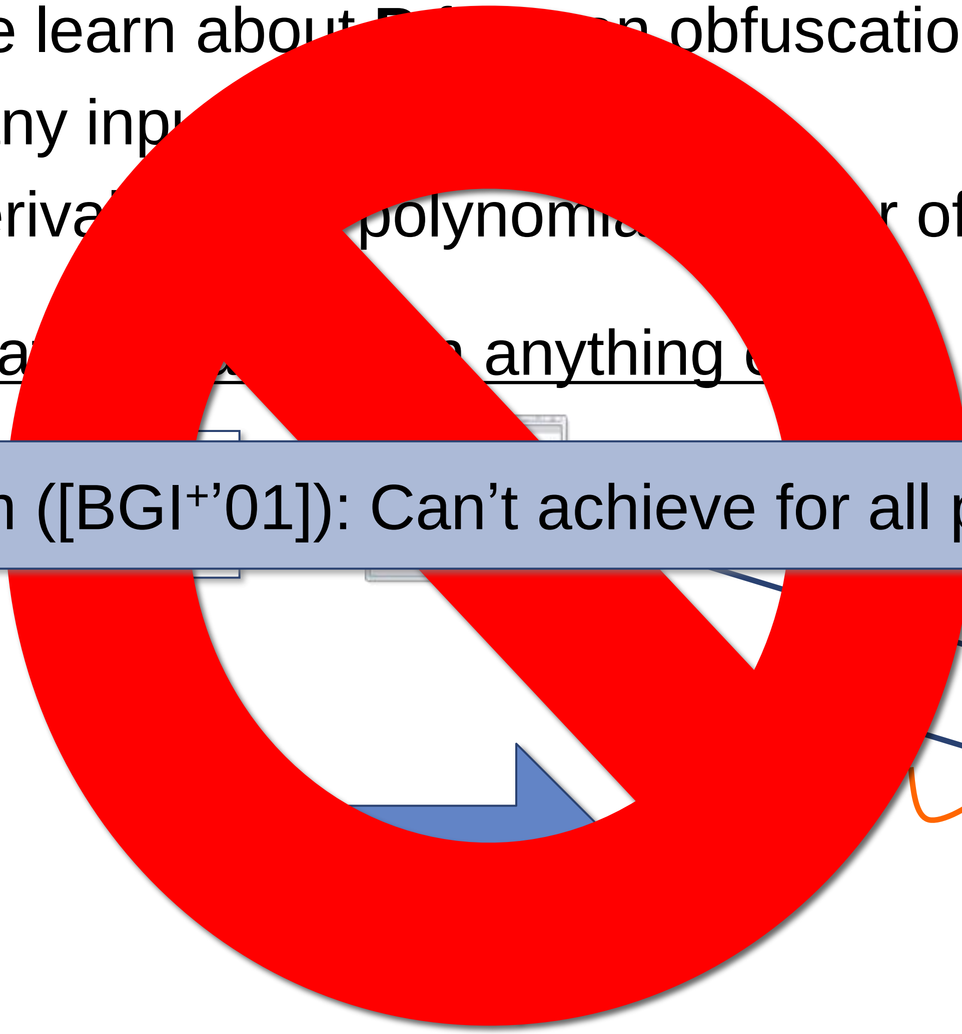
<u>VBB Obfuscation: can't learn anything else</u>



Virtual Black Box (VBB) Obfuscation [BGIRSVY'01]

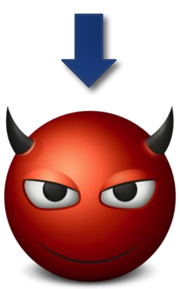
What can we learn abou… obfuscation $\mathbf{P_O}$?

- Output on any inp…
- Anything deriva… polynomia… r of outputs



VBB Obfusca… anything …

Theorem ([BGI+'01]): Can't achieve for all programs



b=0,1

b

More on VBB Impossibility

BGI$^+$ construct program **P** with embedded secret **k** where:

- **k** is secret even given black box access to **P**
- Given <u>any</u> program computing **P**, can recover **k**

Main takeaways:

- Need weaker notion of obfuscation
- Obfuscation alone cannot guarantee secret hiding

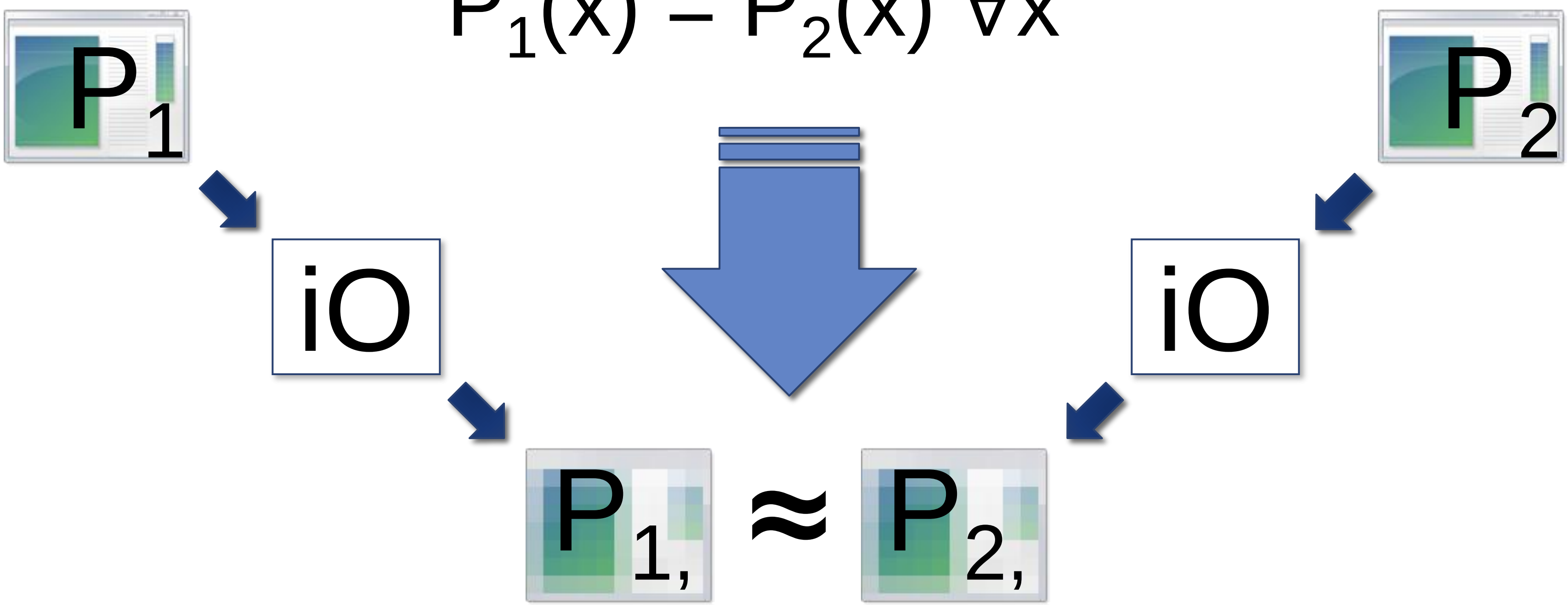
Example:

- Some encryption schemes cannot be obfuscated
- Perhaps specific encryption schemes can be obfuscated?
  - e.g. public key encryption schemes

Indist. Obfuscation (iO) [BGI+'01, GR'07]

If two programs have same functionality, obfuscations are indistinguishable

$$P_1(x) = P_2(x) \ \forall x$$



BGI+ counter example does not apply to iO

However, big questions: How to build? How to use?

Indistinguishability Obfuscation (iO)

Answer:

- **[GGHRSW'13] First candidate iO construction**
  - Functional encryption

Exploding field:

- [BR'13, BGKPS'13, …] Additional constructions
- [SW'13, GGHR'13, **BZ'13**, **ABGSZ'13**, …] Uses
  - Public key encryption, signatures, deniable encryption, multiparty key exchange, MPC, …
- [BCPR'13, MR'13, BCP'13, …] Further Investigation

Our Results

Non-interactive multiparty key exchange

- First scheme without trusted setup

Efficient broadcast encryption

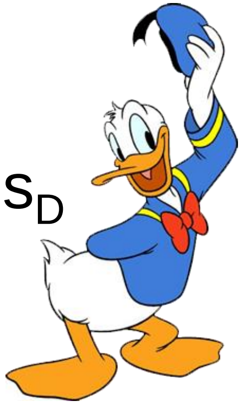
- Constant size ciphertext and secret keys
- First *distributed* system: users generate keys themselves

Efficient traitor tracing

- Shortest secret keys, ciphertexts, known
- Resolves open problem in Differential Privacy [DNR+09]

MULTIPARTY KEY EXCHANGE

(Non-Interactive) Multiparty Key Exchange

s_A  s_B  s_C  s_D 

K_{ABCD} K_{ABCD} K_{ABCD} K_{ABCD}

History

2 parties: Diffie Hellman Protocol [DH’76]

3 parties: Bilinear maps [Joux’2000]

n>3 parties: Multilinear maps [BS’03,GGH’13,CLT’13]

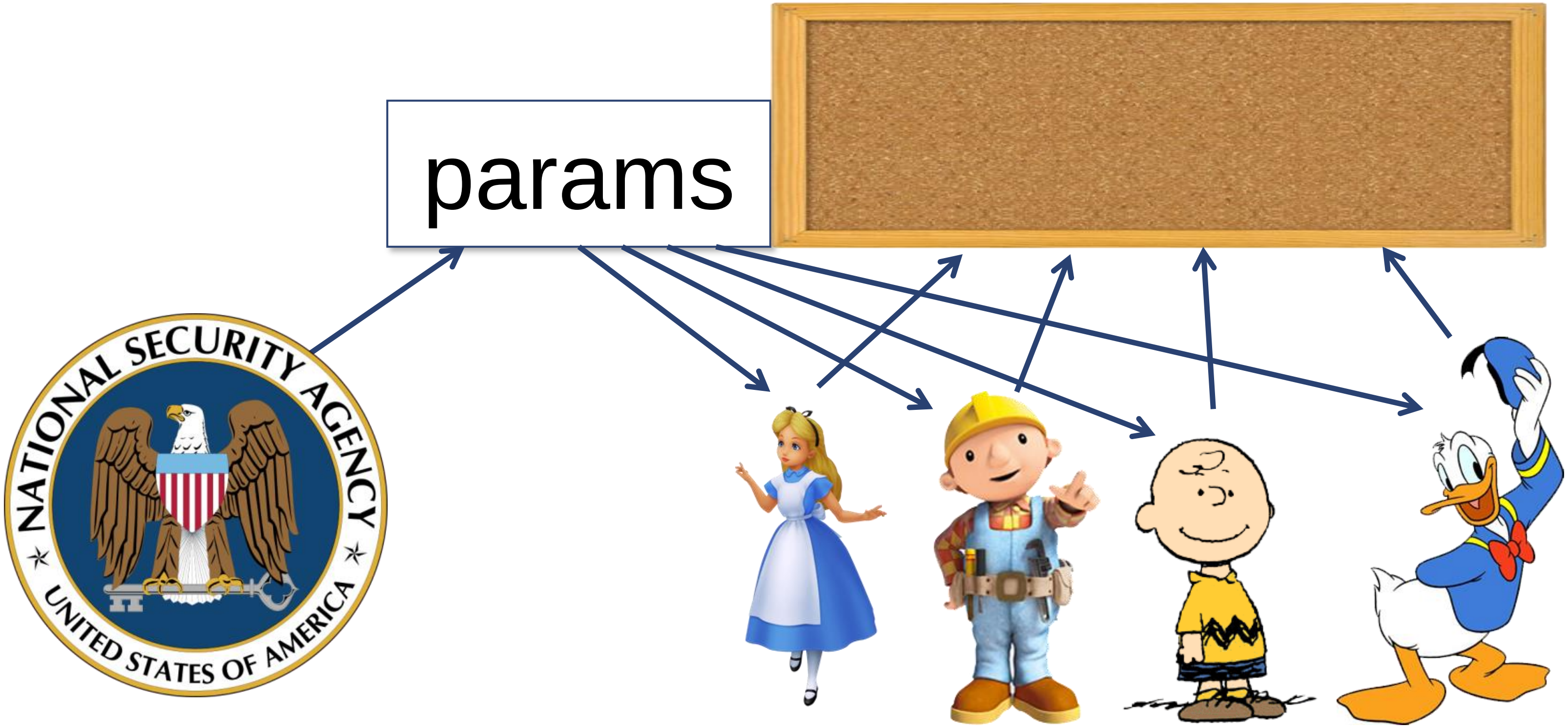
- Requires trusted setup phase

Our work: **n** parties, no trusted setup

Prior Constructions for **n>3**

First achieved using multilinear maps [GGH'13,CLT'13]

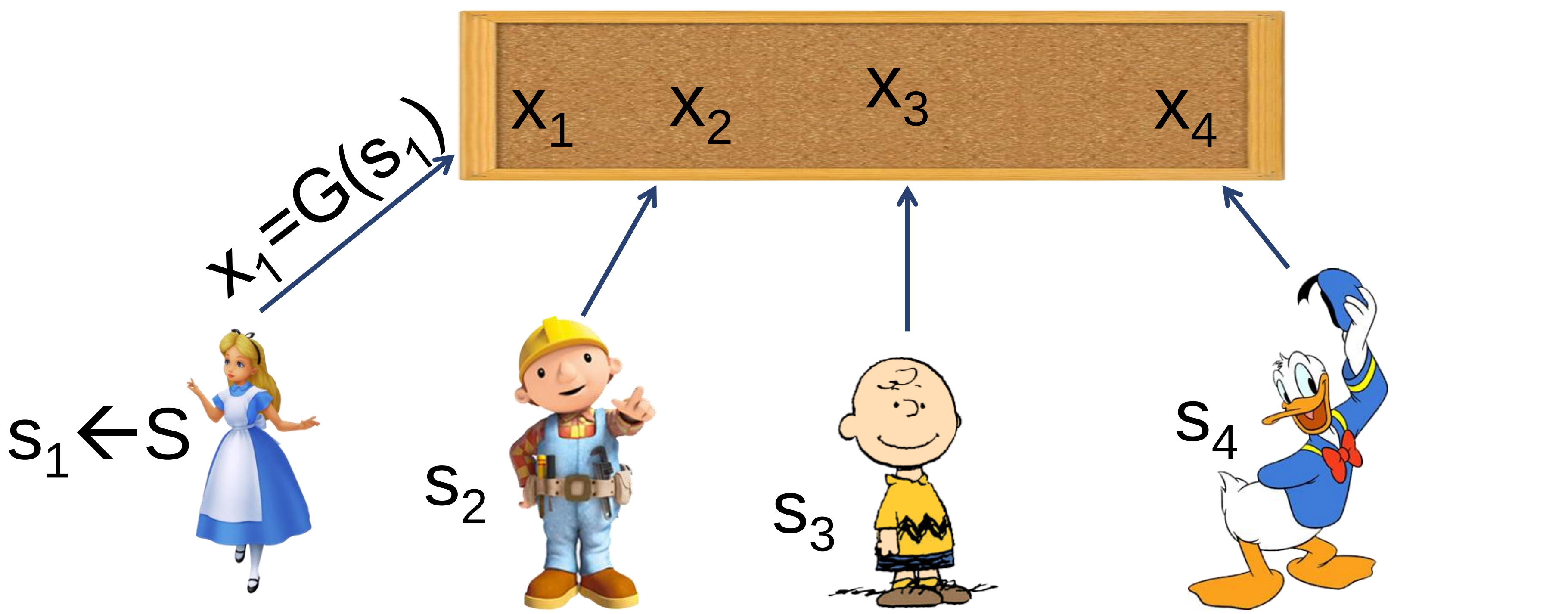
- These constructions all require trusted setup **before** protocol is run
- Trusted authority can also learn group key



Starting point for our construction

Building blocks:

- One-way function **G:S → X**
- Pseudorandom function (PRF) **F**



Shared key: $F_k(x_1, x_2, x_3, x_4)$ ← how to compute securely?

Introduce Trusted Authority (for now)

k

$P(x'_1, \ldots, x'_n, s, i)$ {
  If $G(s) \neq x'_i$, output $\perp$
  Otherwise, output $F_k(x'_1, \ldots, x'_n)$
}

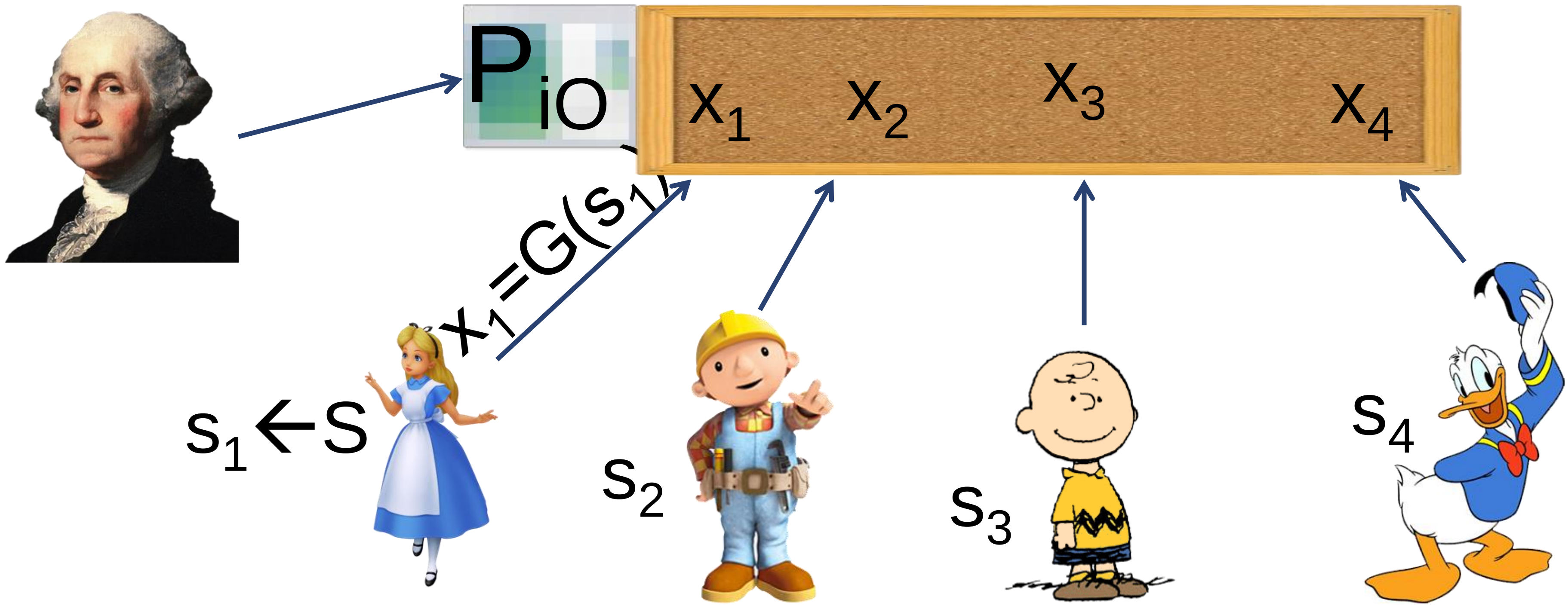
↓

iO

↓

P_i

First attempt



$$K_{ABCD} = P_{iO}(x_1, x_2, x_3, x_4, s_1, 1)$$

Problems:

- **k** not guaranteed to be hidden using iO
- Still have trusted authority

Removing Trusted Setup

As described, our scheme needs trusted setup

Observation: Obfuscated program can be generated independently of publishing step

k	P(x'_1, ..., x'_n, s, i) {
	     If $G(s) \neq x'_i$, output $\perp$
	     Otherwise, output $F_k(x'_1, \ldots, x'_n)$
	 }

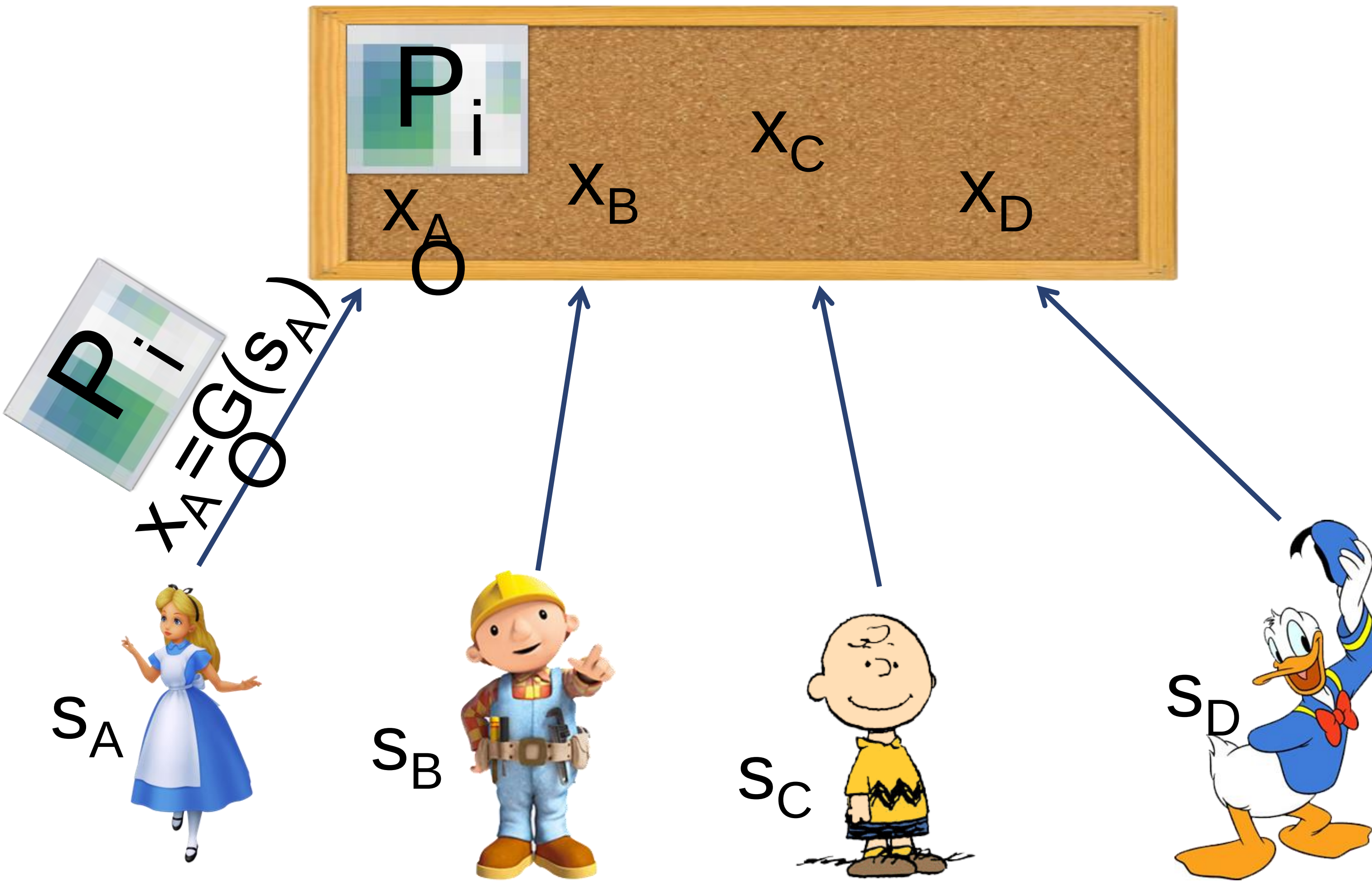
→ iO →

O

Untrusted setup: designate user 1 as “master party”

- generates $\mathbf{P_{iO}}$, sends with $\mathbf{x_1}$

Multiparty Key Exchange Without Trusted Setup



Security equivalent to security of previous scheme

Hiding **k**

Follow "punctured program" paradigm of SW'13

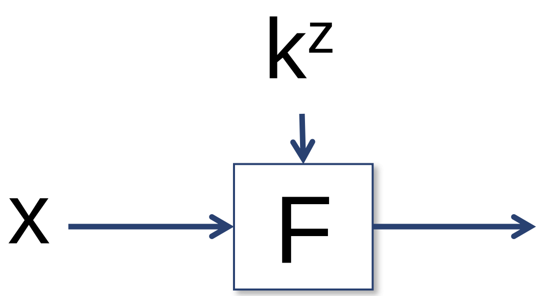
- Use pseudorandom generator for **G**

  G: S → X |X| >> |S|

  G(s), s←S indist. from **x←X**

- Use special "punctured PRF" for **F** [BW'13, KPTZ'13, BGI'13, SW'13]

  Punctured key $\mathbf{k^z}$ ⇒ compute $\mathbf{F_k(\cdot)}$ everywhere but **z**

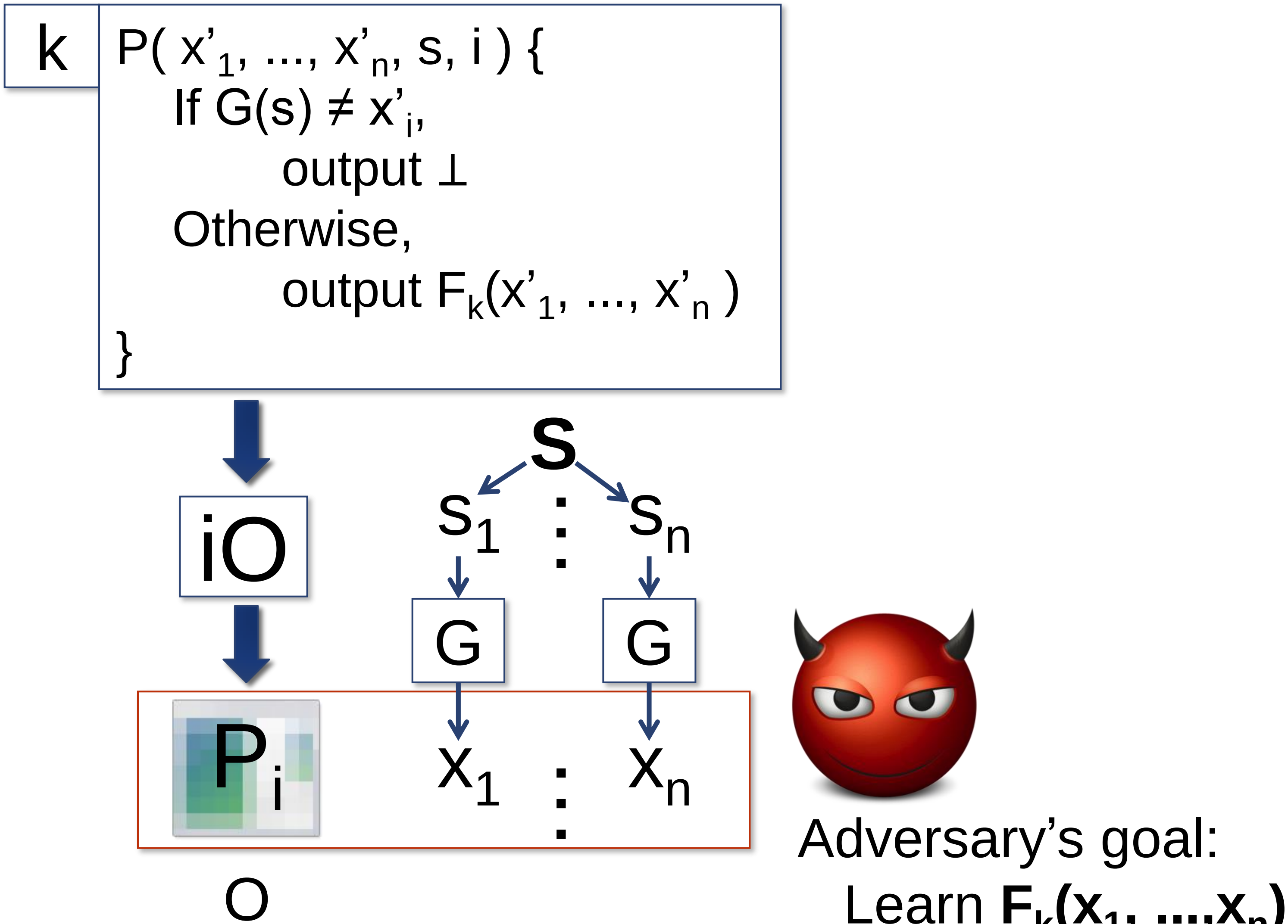


F(k,x)   if x ≠ z
⊥   if x = z

Security: given $\mathbf{k^z}$, cannot compute $\mathbf{t=F_k(z)}$

Construction: GGM'84

Security of Our Construction

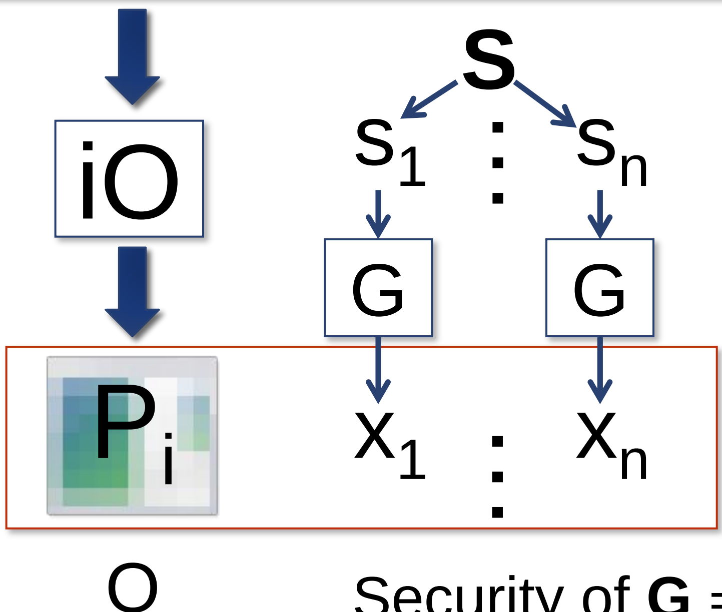


Step 1: Replace $\mathbf{x_i}$

Real World

k

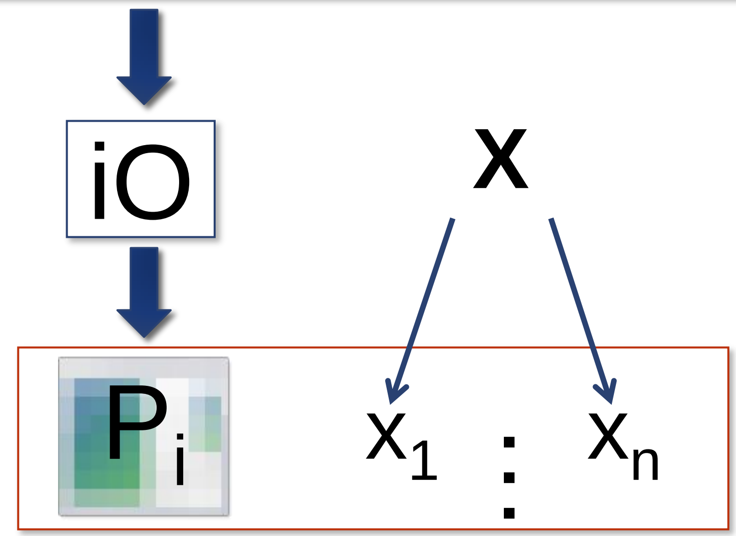
P(x'_1, ..., x'_n, s, i) {
  If G(s) ≠ x'_i,
    output ⊥
  Otherwise,
    output $F_k(x'_1, \ldots, x'_n)$
}



Alternate World 1

k

P(x'_1, ..., x'_n, s, i) {
  If G(s) ≠ x'_i,
    output ⊥
  Otherwise,
    output $F_k(x'_1, \ldots, x'_n)$
}



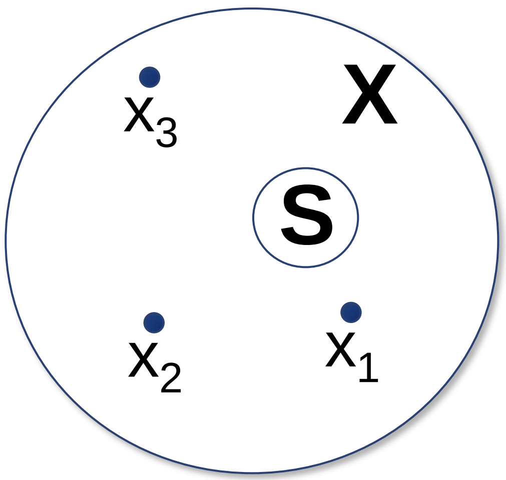
Security of **G** ⇒ words indistinguishable

Step 1: Replace $\mathbf{x_i}$

Observation:

Since **|X| >> |S|**,

w.h.p. no **s,i** s.t. $\mathbf{G(s)=x_i}$



Never pass check when

$\mathbf{x'_1, \ldots, x'_n = x_1, \ldots, x_n}$

<u>Alternate World 1</u>

k

P(x'_1, ..., x'_n, s, i) {

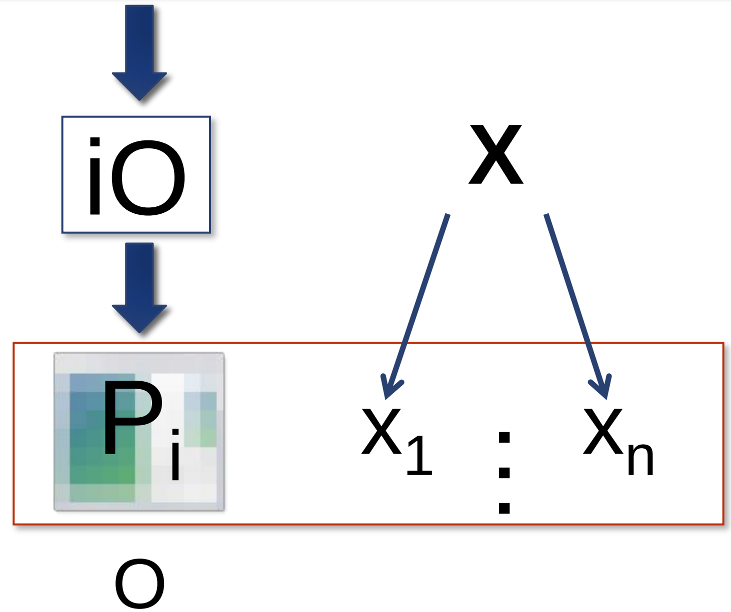
If G(s) ≠ x'_i,

output ⊥

Otherwise,

output $F_k(x'_1, \ldots, x'_n)$

}



Step 2: Puncture

Alternate World 2

k^z

P(x'_1, ..., x'_n, s, i) {
    If G(s) ≠ x'_i,
        output ⊥
    If (x'_1, ..., x'_n) = z,
        output ⊥
    Otherwise,
        output $F_k(x'_1, ..., x'_n)$
}

iO

P_i

X

x_1 : x_n

Let **z=(x_1, ..., x_n)**

Alternate World 1

k

P(x'_1, ..., x'_n, s, i) {
    If G(s) ≠ x'_i,
        output ⊥
    Otherwise,
        output $F_k(x'_1, ..., x'_n)$
}

iO

P_i

X

x_1 : x_n

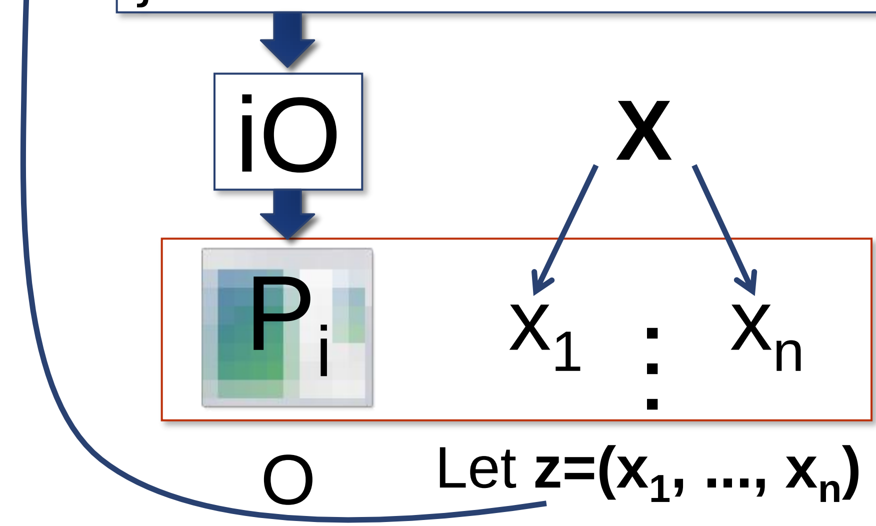
W.h.p. programs identical + iO ⇒ Worlds indistinguishable

Security

<u>Alternate World 2</u>

k^z

P($x'_1, \ldots, x'_n$, s, i) {
  If $G(s) \neq x'_i$,
    output ⊥
  If $(x'_1, \ldots, x'_n) = z$,
    output ⊥
  Otherwise,
    output $F_k(x'_1, \ldots, x'_n)$
}



Adversary's goal: learn $\mathbf{F_k(z)}$

Success in Real World
⇒ success in World 2

In World 2:
Adversary only sees $\mathbf{k^z}$
⇒ cannot learn $\mathbf{F_k(z)}$

✓

Minimal Assumptions

Building blocks:

- iO
- Pseudorandom generator **G: S** $\rightarrow$ **X** (**|X| >> |S|**)
- Puncturable PRF **F: K × X**$^{\mathbf{n}}$ $\rightarrow$ **Y**

[GGM’84] [HILL’99]

iO + NP $\not\subseteq$ Co-RP $\xrightarrow{\text{[MR'13]}}$ One-way functions

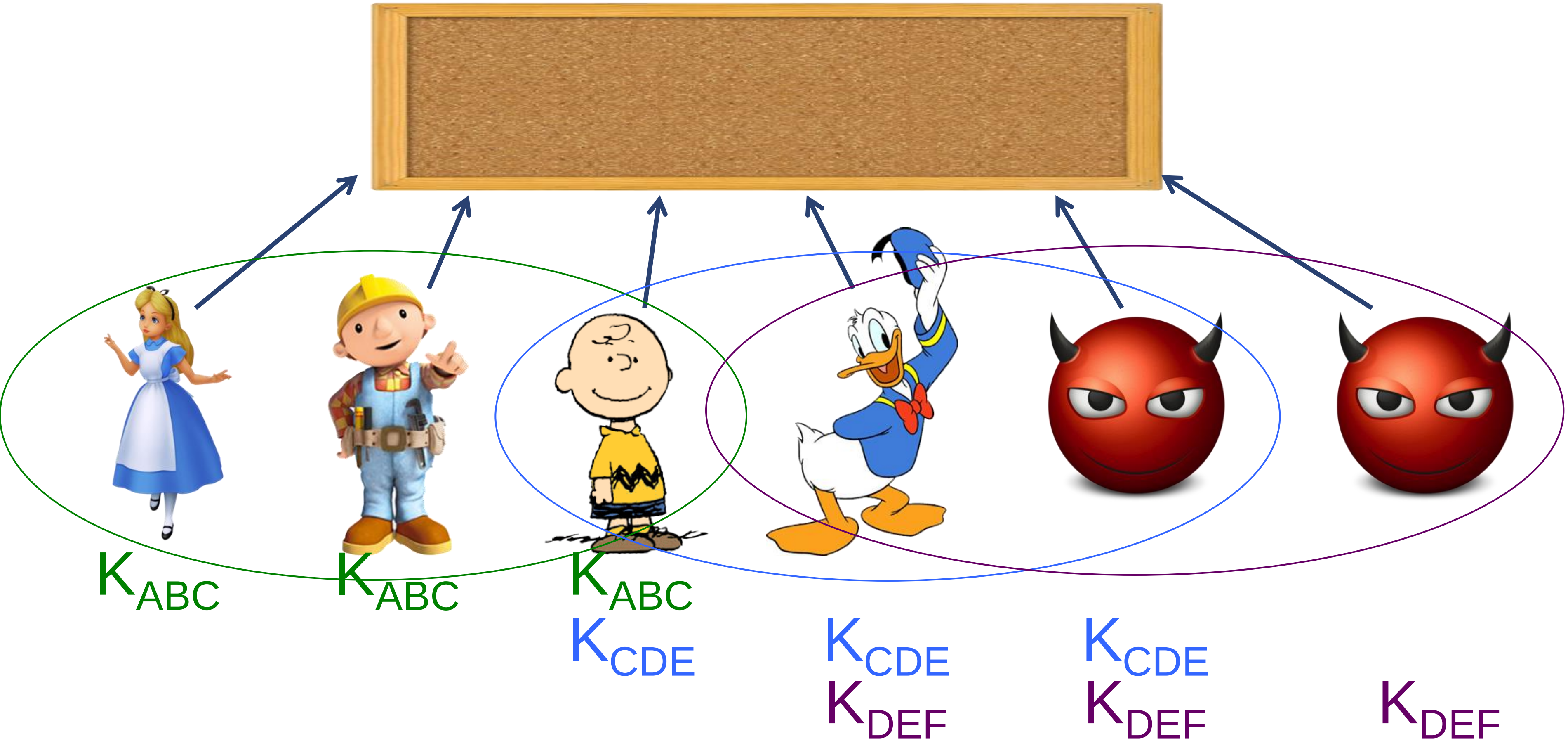
Our constructions can be built from iO and worst-case complexity assumptions

ACTIVE SECURITY

Active Notions of Security

Key exchange protocol may be used multiple times

- Adversary may take part as well (even multiple times)



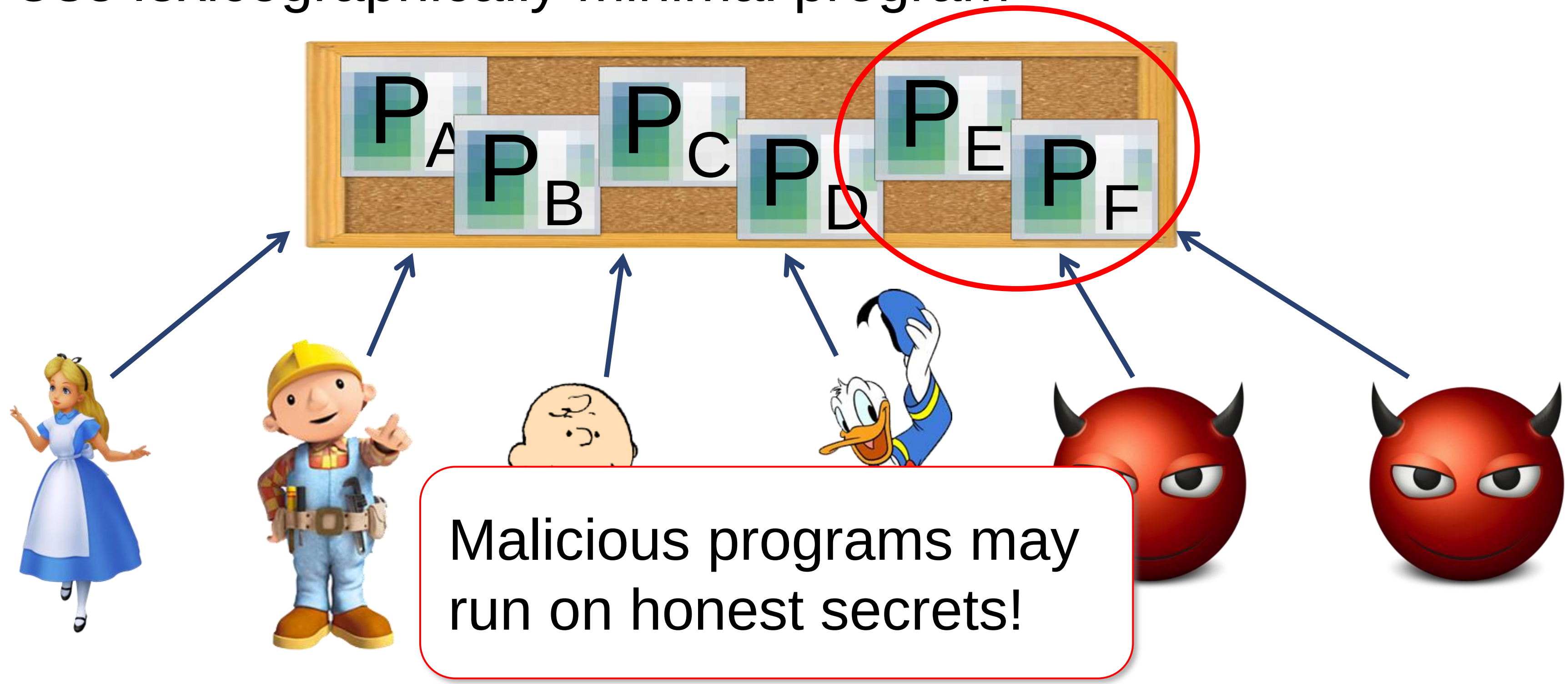
Active Notions of Security

Implications for our scheme:

- Everyone must be ready to be “master party”

  ⇒ everyone must publish own program $\mathbf{P_i}$

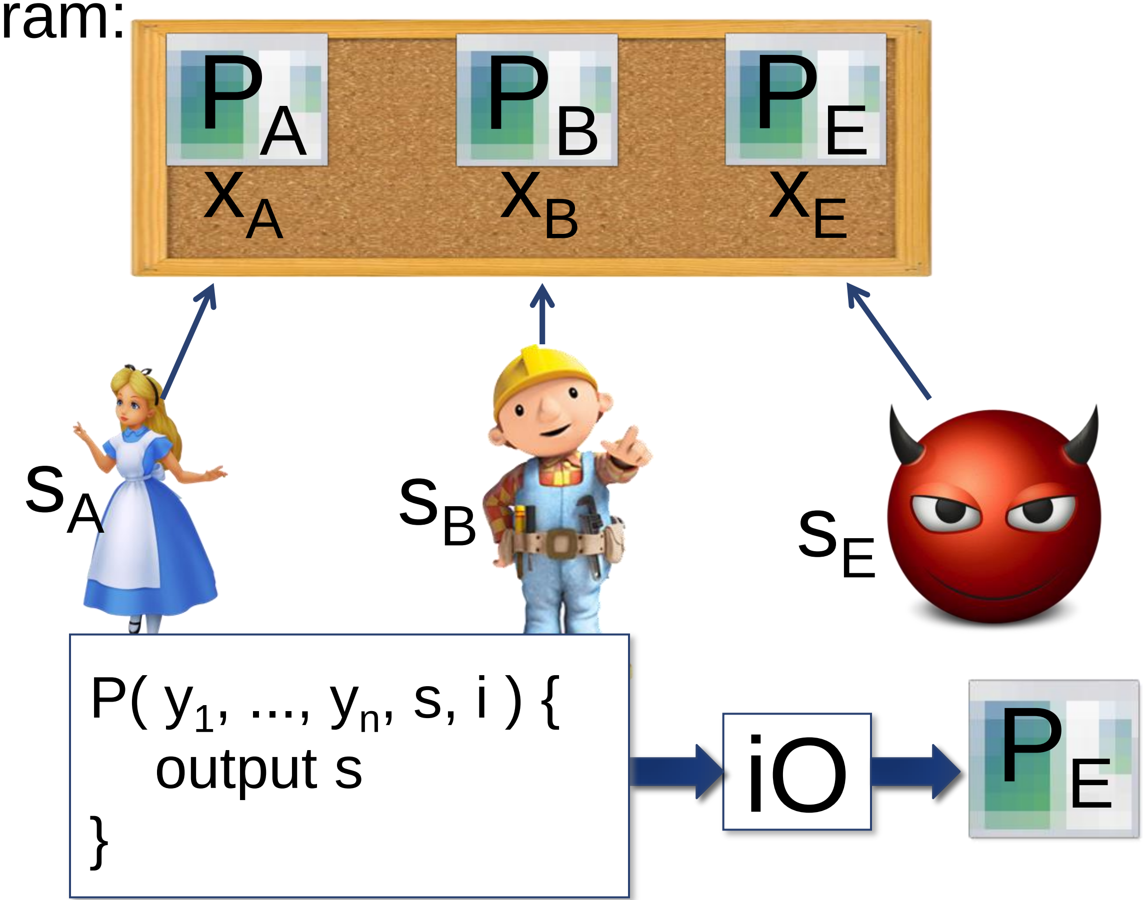
- Use lexicographically minimal program



Active Notions of Security

Potential attack:

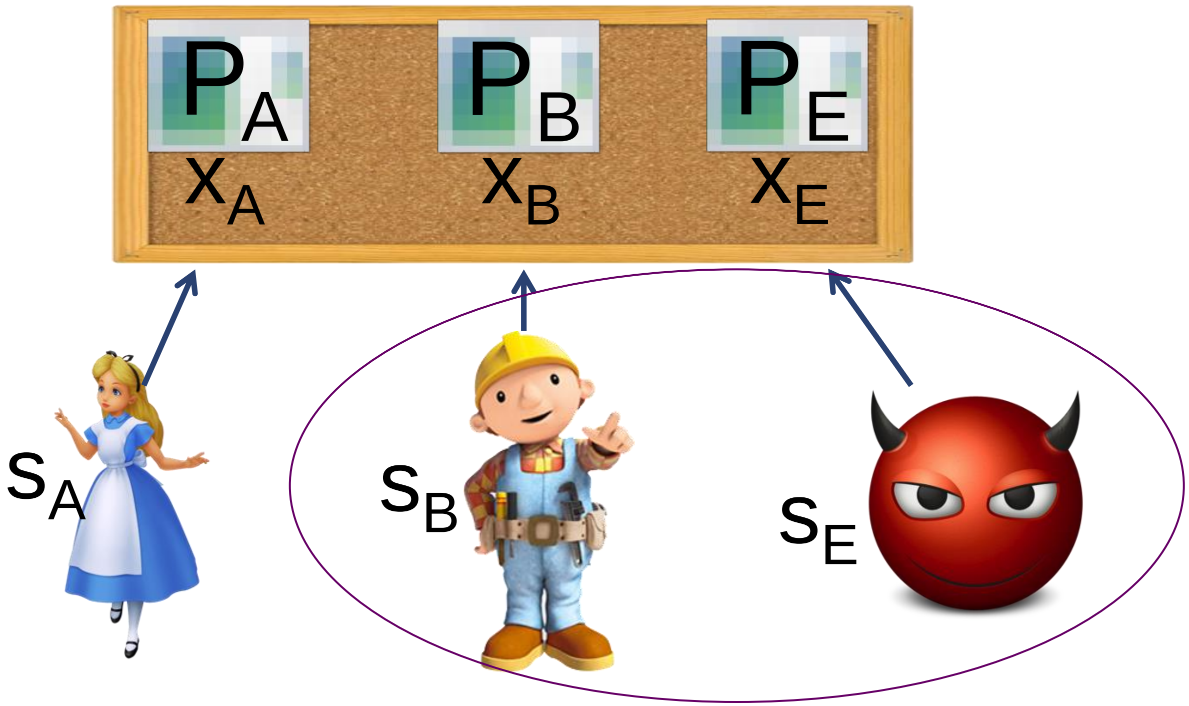
- Step 1: Attacker creates and publishes malicious program:



Active Notions of Security

Potential attack:

- Step 2: Attacker and Bob use attacker's program:

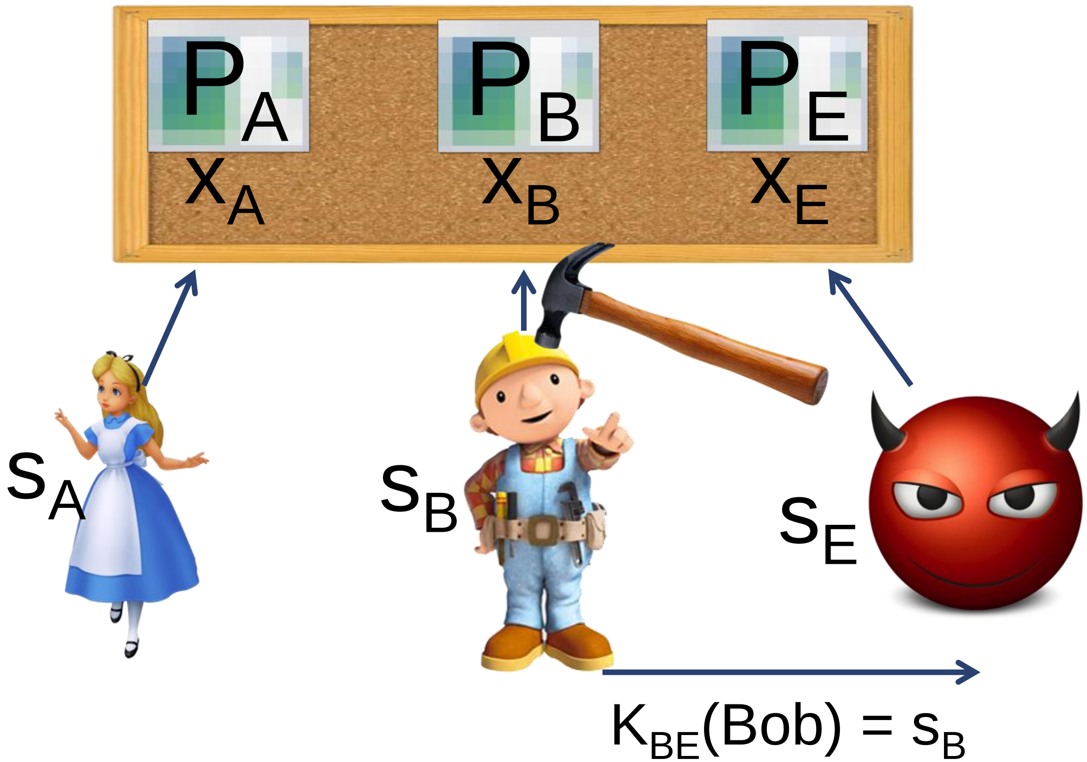


$K_{BE}(\text{Bob}) = P_E(x_B, x_E, s_B, B) = s_B$

Active Notions of Security

Potential attack:

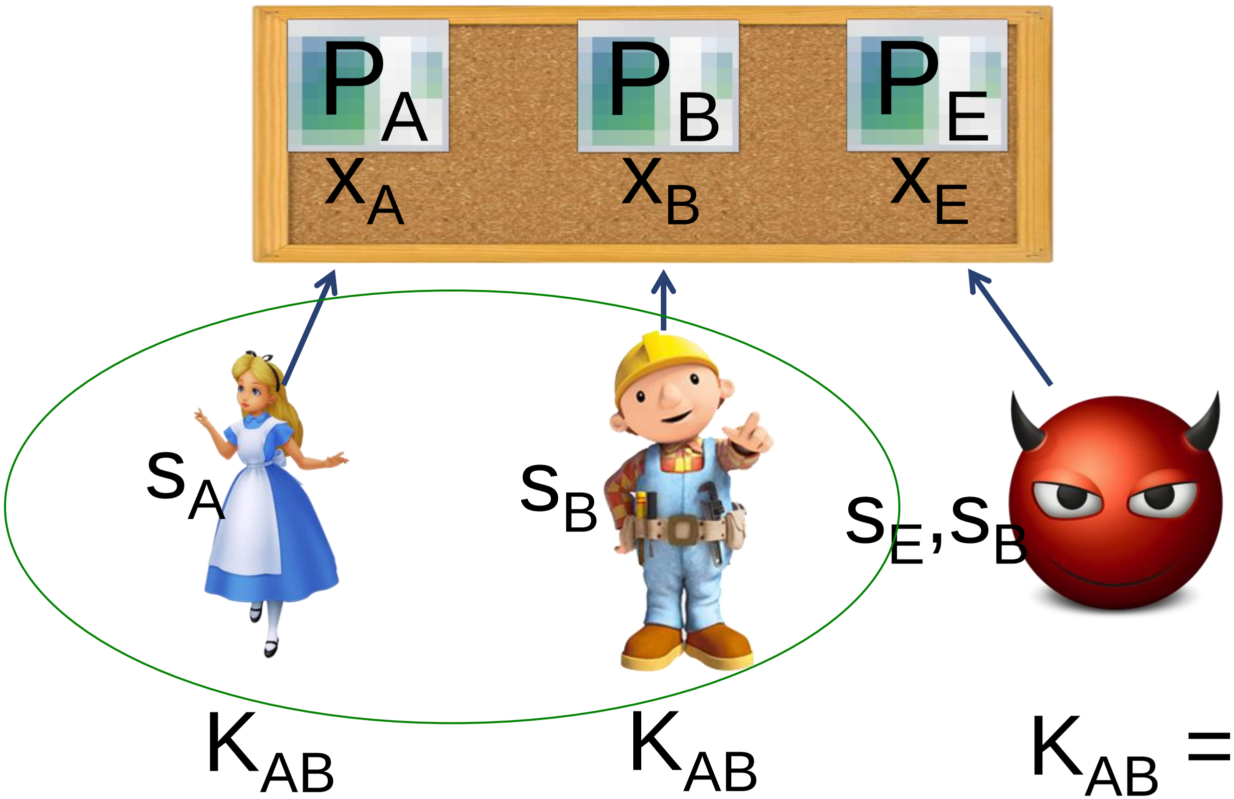
- Step 3: Attacker steals Bob's shared keys:



Active Notions of Security

Potential attack:

- Step 4: Attacker can compute any future shared key:



$K_{AB} = P_A(x_A, x_B, s_B, B)$

Problems with Basic Scheme

Malicious programs run on honest secrets

Ways to fix?

- Ensure programs are honest

  Problematic since program obfuscated

- Never run untrusted programs on secrets

  (Assume inputs to completely leak)

Our Solution

- Replace user secret with signing key for signature scheme
- Publish public key
- Input to program is signature on set of users

F

```
P( pk_1, ..., pk_n, S, σ, i ) {
    If Ver( pk_i, S, σ ) rejects, output ⊥
    Otherwise, output F(k, pk_1, ..., pk_n)
}
```

Intuition: Even after seeing many signatures, cannot learn signature on challenge set

Theorem: iO + "constrained signature" + "constrained PRF" ⇒ "semi-static" security

Build from iO

Intermediate sec. notion

[BW'13]: build from MLM
Or, build from iO

REDUCING PARAMETER SIZES

Reducing Parameter Sizes [ABGSZ'13]

Key exchange program:

k	P(x'_1, ..., x'_n, s, i) {
	    If $G(s) \neq x'_i$, output $\perp$
	    Otherwise, output $F(k, x'_1, ..., x'_n)$
	}

Size of input: $\mathbf{\Omega(n)}$

For circuits, size of program: $\mathbf{\Omega(n)}$

- Also true for Turing Machines (less obvious)

To reduce program size, must reduce input size

$\Rightarrow$ Must derive key from small string

Reducing Parameter Sizes

Idea: use hash of public values to derive key

$$h \leftarrow H(x_1, \ldots, x_n)$$
$$k \leftarrow F(k,h)$$

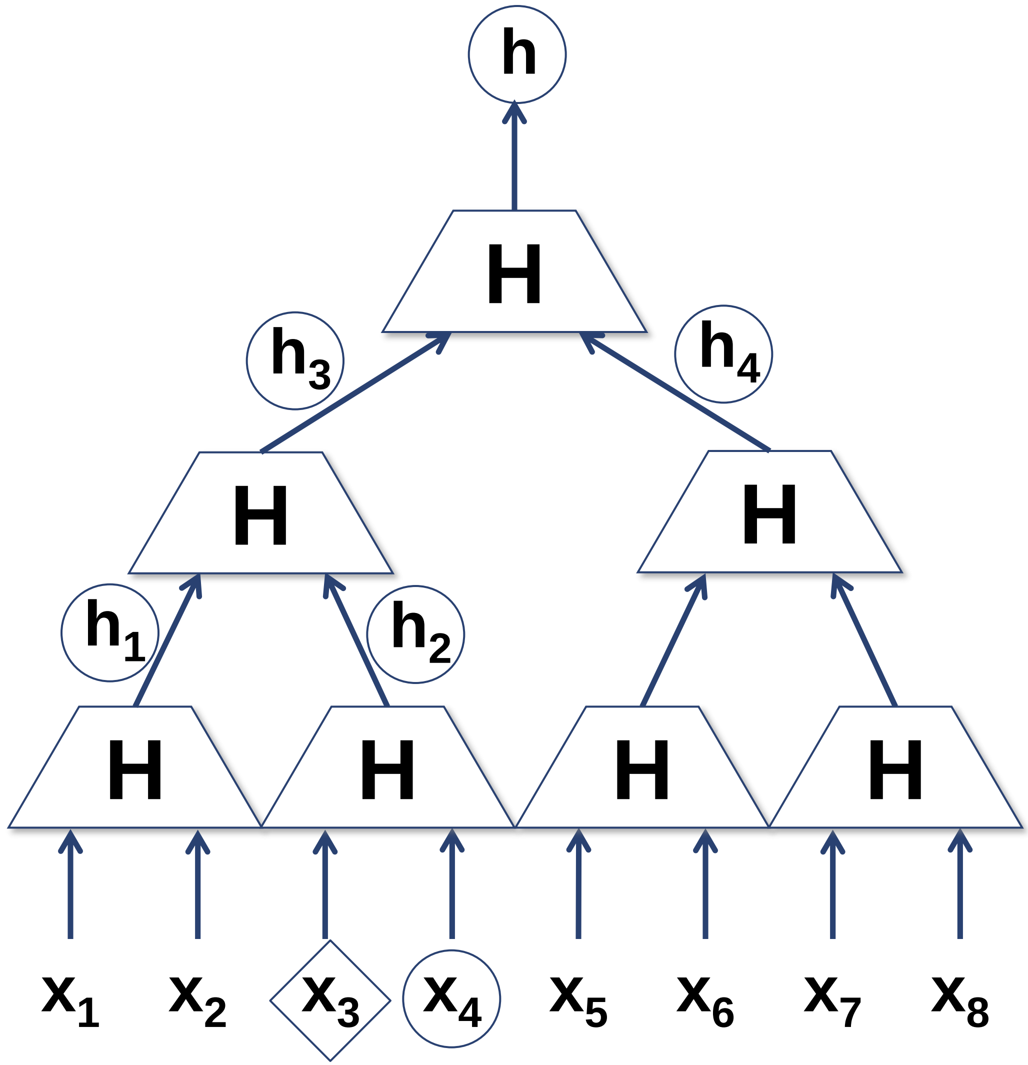
User supplies **h** to program

Question: How does user **i** prove **h** is correct?

- Need proof that $\mathbf{h=H(x'_1, \ldots, x'_n)}$ where $\mathbf{x_i' = x_i}$
- Need proof to be small

Answer: Merkle Hash Trees

Merkle Hash Trees



Proof size: **O(log n)**

Our Construction

k	P(h, π, x, s, i) {
	    If π is an invalid proof for (h,x,i), output ⊥
	    If G(s) ≠ x, output ⊥
	    Otherwise, output F(k,h)
	}

```
P( h, π, x, s, i ) {
    If π is an invalid proof for (h,x,i), output ⊥
    If G(s) ≠ x, output ⊥
    Otherwise, output F(k,h)
}
```

Program size: **poly(log n)**

Problem: false proofs exist (though hard to compute)

- Must use stronger notion of obfuscation: diO

Open Questions

Reduce program sizes using iO?

Other primitives from iO

- FHE?
- Multilinear maps?

Thanks!