Indistinguishability Obfuscation via Mathematical Proofs of Equivalence

Abhishek Jain

Johns Hopkins University

Zhengzhong Jin

Johns Hopkins University → MIT

Indistinguishability Obfuscation (iO)

```
function main() {
  console.log('hello, world');
}
main()
```

Source code

iO →

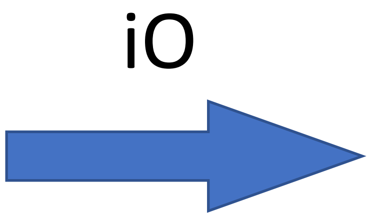
```
function _0x19e6(_0x4d301f,_0xcaab53){var _0x3a4e72=_0x3a4e();return
_0x19e6=function(_0x19e691,_0x5809f0){_0x19e691=_0x19e691-0x14e;var
_0x16ee0b=_0x3a4e72[_0x19e691];return
_0x16ee0b;},_0x19e6(_0x4d301f,_0xcaab53);}function _0x3a4e(){var _0x3f0a9d=
['log','199381NCGrSa','2328491tAiNSg','18mVqyqS','4cVQTsk','6PuGzwR','107410
32WsiTVO','104321yYIIVM','370911DTLqdw','10uRQffV','2024504eEkwnt','114dOcOh
j','hello,\x20world','2634710IatlOd'];_0x3a4e=function(){return
_0x3f0a9d;};return _0x3a4e();}(function(_0x3d9e47,_0x360e03){var
_0x3afd0b=_0x19e6,_0x2928d3=_0x3d9e47();while(!![]){try{var _0x33cc3a=-
parseInt(_0x3afd0b(0x15a))/0x1*(-parseInt(_0x3afd0b(0x158))/0x2)+-
parseInt(_0x3afd0b(0x15b))/0x3*(-parseInt(_0x3afd0b(0x157))/0x4)+-
parseInt(_0x3afd0b(0x152))/0x5+parseInt(_0x3afd0b(0x150))/0x6*
(parseInt(_0x3afd0b(0x154))/0x7)+-parseInt(_0x3afd0b(0x14f))/0x8*(-
parseInt(_0x3afd0b(0x156))/0x9)+parseInt(_0x3afd0b(0x14e))/0xa*
(parseInt(_0x3afd0b(0x155))/0xb)+-
parseInt(_0x3afd0b(0x159))/0xc;if(_0x33cc3a===_0x360e03)break;else
_0x2928d3['push'](_0x2928d3['shift']());}catch(_0x437e27){_0x2928d3['push']
(_0x2928d3['shift']());}}}(_0x3a4e,0x42c94));function main(){var
_0x29ace6=_0x19e6;console[_0x29ace6(0x153)](_0x29ace6(0x151));}main();
```

"Unintelligible"

Indistinguishability Obfuscation (iO)

```
function main() {
  console.log('hello, world');
}
main()
```

Source code



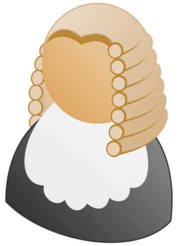

```
function _0x19e6(_0x4d301f,_0xcaab53){var _0x3a4e72=_0x3a4e();return
_0x19e6=function(_0x19e691,_0x5809f0){_0x19e691=_0x19e691-0x14e;var
_0x16ee0b=_0x3a4e72[_0x19e691];return
_0x16ee0b;},_0x19e6(_0x4d301f,_0xcaab53);}function _0x3a4e(){var _0x3f0a9d=
['log','199381NCGrSa','2328491tAiNSg','18mVqyqS','4cVQTsk','6PuGzwR','107410
32WsiTVO','104321yYIIVM','370911DTLqdw','10uRQffV','2024504eEkwnt','114dOcOh
j','hello,\x20world','2634710IatlOd'];_0x3a4e=function(){return
_0x3f0a9d;};return _0x3a4e();}(function(_0x3d9e47,_0x360e03){var
_0x3afd0b=_0x19e6,_0x2928d3=_0x3d9e47();while(!![]){try{var _0x33cc3a=-
parseInt(_0x3afd0b(0x15a))/0x1*(-parseInt(_0x3afd0b(0x158))/0x2)+-
parseInt(_0x3afd0b(0x15b))/0x3*(-parseInt(_0x3afd0b(0x157))/0x4)+-
parseInt(_0x3afd0b(0x152))/0x5+parseInt(_0x3afd0b(0x150))/0x6*
(parseInt(_0x3afd0b(0x154))/0x7)+-parseInt(_0x3afd0b(0x14f))/0x8*(-
parseInt(_0x3afd0b(0x156))/0x9)+parseInt(_0x3afd0b(0x14e))/0xa*
(parseInt(_0x3afd0b(0x155))/0xb)+-
parseInt(_0x3afd0b(0x159))/0xc;if(_0x33cc3a===_0x360e03)break;else
_0x2928d3['push'](_0x2928d3['shift']());}catch(_0x437e27){_0x2928d3['push']
(_0x2928d3['shift']());}}}(_0x3a4e,0x42c94));function main(){var
_0x29ace6=_0x19e6;console[_0x29ace6(0x153)](_0x29ace6(0x151));}main();
```

"Unintelligible"

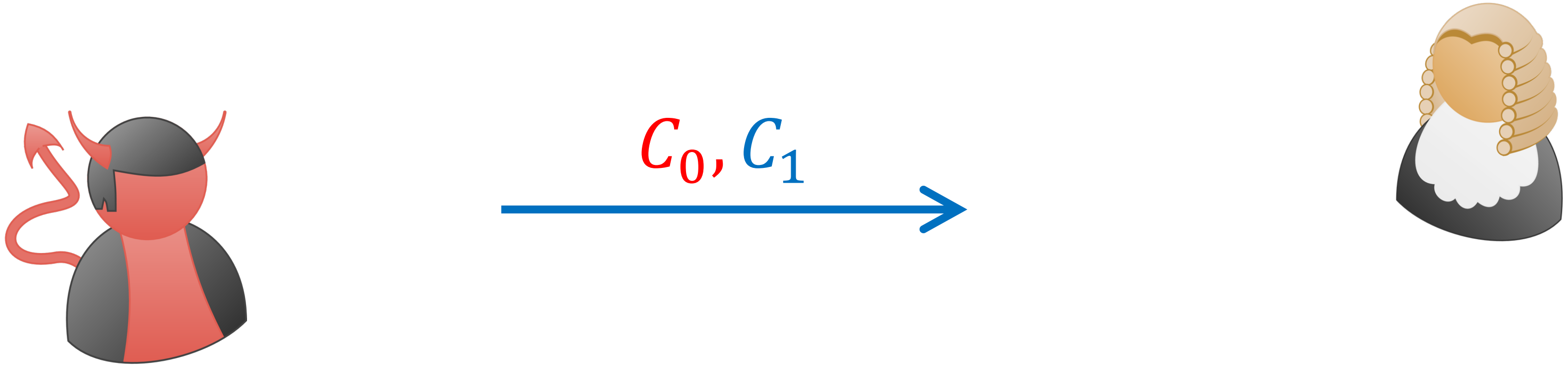
The obfuscated program *preserves the functionality* of the input program.

(Produce the same output)

Indistinguishability Security (as a Game)

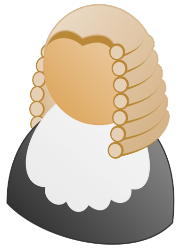


Indistinguishability Security (as a Game)



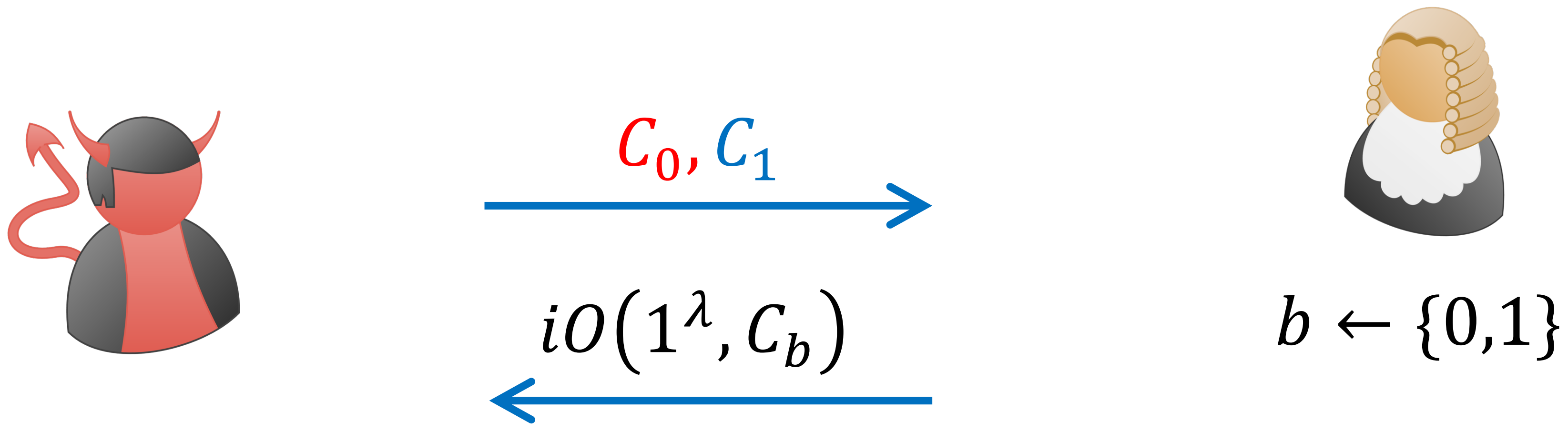
Indistinguishability Security (as a Game)

C_0, C_1 →



$b \leftarrow \{0,1\}$

Indistinguishability Security (as a Game)

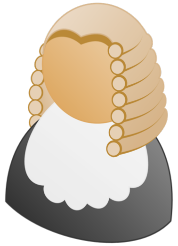


Indistinguishability Security (as a Game)

C_0, C_1 →

← $iO(1^\lambda, C_b)$

b' →



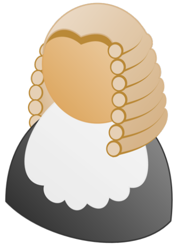
$b \leftarrow \{0,1\}$

Indistinguishability Security (as a Game)

C_0, C_1 →

← $iO(1^\lambda, C_b)$

b' →



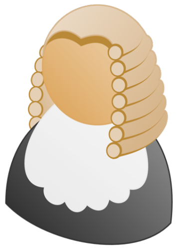
$b \leftarrow \{0,1\}$

If $\forall\ x\ C_0(x) = C_1(x)$ and $b = b'$, adversary wins.

Indistinguishability Security (as a Game)

C_0, C_1 →

$iO(1^\lambda, C_b)$ ←



$b \leftarrow \{0,1\}$

b' →

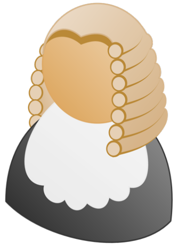
If $\forall\, x\; C_0(x) = C_1(x)$ and $b = b'$,
adversary wins.

$\Pr[$  $wins] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$

Indistinguishability Security (as a Game)

C_0, C_1 →

← $iO(1^\lambda, C_b)$



$b \leftarrow \{0,1\}$

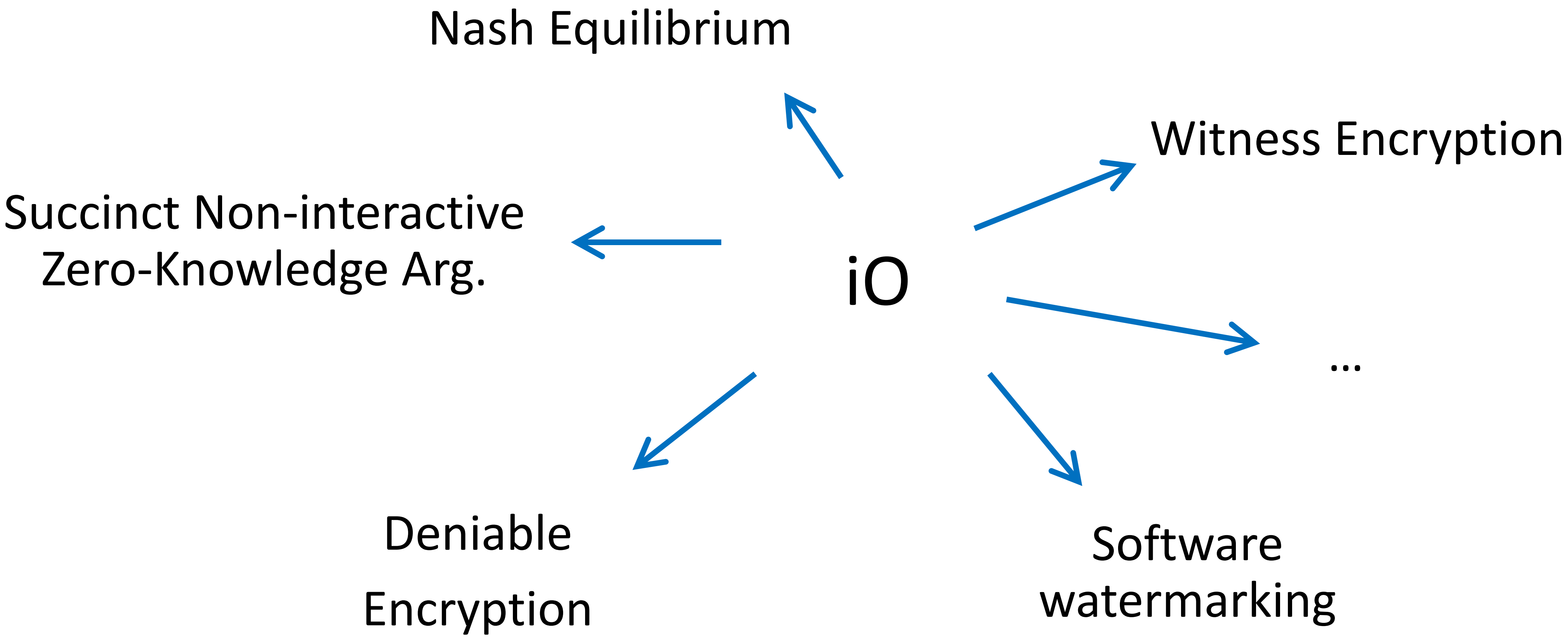
b' →

Important!

If $\forall\, x\ C_0(x) = C_1(x)$ and $b = b'$, adversary wins.

$\Pr[$  $wins] \le \frac{1}{2} + \mathrm{negl}(\lambda)$

iO: Crypto "Complete" [Sahai-Waters'13,...]



Can we build iO?

Can we build iO?

A Long Line of Work:

[Garg-Gentry-Halevi-Raykova-Sahai-Waters’13][Pass-Seth-Telang’14]
[Gentry-Lewko-Sahai-Waters’15][Ananth-Jain’15][Bitansky-Vaikuntanathan’15]
[Lin’16][Lin-Vaikuntanathan’16][Lin-Pass-Karn Seth-Telang’16]
[Garg-Miles-Mukherjee-Sahai-Srinivasan-Zhandry’16][Ananth-Sahai’17][Lin’17]
[Lin-Tessaro’17][Agrawal’19][Jain-Lin-Matt-Sahai’19][Brakerski-Dottling-Malavolta’20]…

Can we build iO?

<u>A Long Line of Work:</u>

[Garg-Gentry-Halevi-Raykova-Sahai-Waters’13][Pass-Seth-Telang’14]
[Gentry-Lewko-Sahai-Waters’15][Ananth-Jain’15][Bitansky-Vaikuntanathan’15]
[Lin’16][Lin-Vaikuntanathan’16][Lin-Pass-Karn Seth-Telang’16]
[Garg-Miles-Mukherjee-Sahai-Srinivasan-Zhandry’16][Ananth-Sahai’17][Lin’17]
[Lin-Tessaro’17][Agrawal’19][Jain-Lin-Matt-Sahai’19][Brakerski-Dottling-Malavolta’20]…

iO for ***circuits*** from well-founded assumptions

[Jain-Lin-Sahai’20]

Question: Can we build iO for **Turing machines**?

Question: Can we build iO for **Turing machines**?

Why Turing machines?

Question: Can we build iO for **Turing machines**?

Why Turing machines?

- Natural representation of programs

Question: Can we build iO for **Turing machines**?

Why Turing machines?

- Natural representation of programs

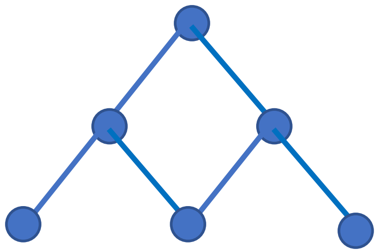
```
function main() {
  console.log('hello, world');
}
main()
```

(Turing Machine)

Question: Can we build iO for **Turing machines**?

Why Turing machines?

- Natural representation of programs
- Support *any* input length



Circuit Model: input length is fixed

Question: Can we build iO for **Turing machines**?

Why Turing machines?

- Natural representation of programs
- Support *any* input length
- *Small* obfuscated program size

```
function main() {
  console.log('hello, world');
}
main()
```

Obfuscated Turing Machine Size:
Poly(input Turing Machine)

Prior Work: Only for *Bounded*-Input Length

[BGLPT'15][CHJV'15][KLW'15][GS'18]…

Prior Work: Only for *Bounded*-Input Length

[BGLPT'15][CHJV'15][KLW'15][GS'18]...

Adversary for iO

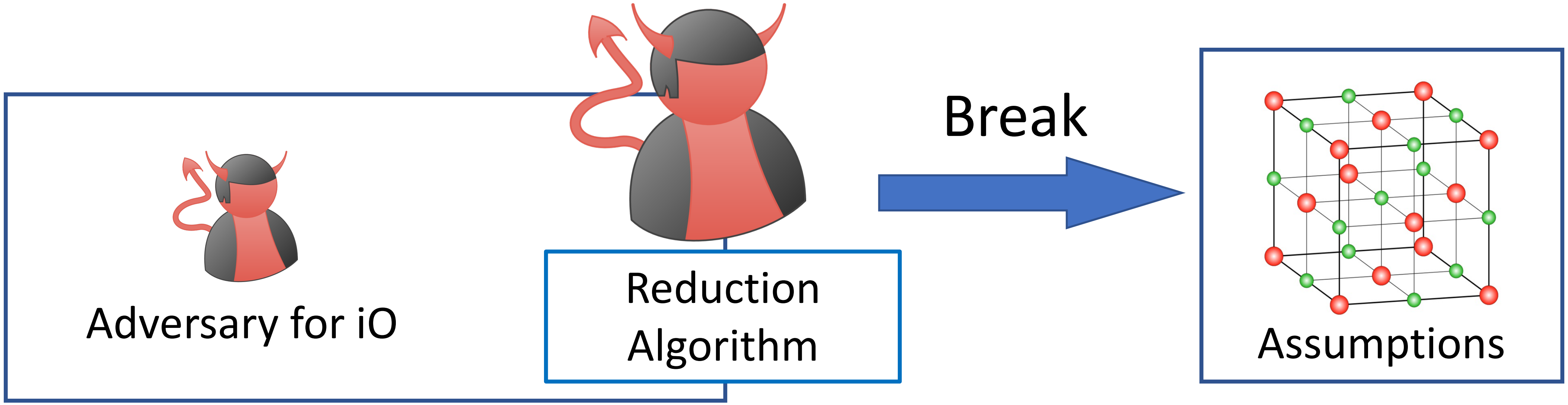
Prior Work: Only for *Bounded*-Input Length

[BGLPT’15][CHJV’15][KLW’15][GS’18]…

Adversary for iO

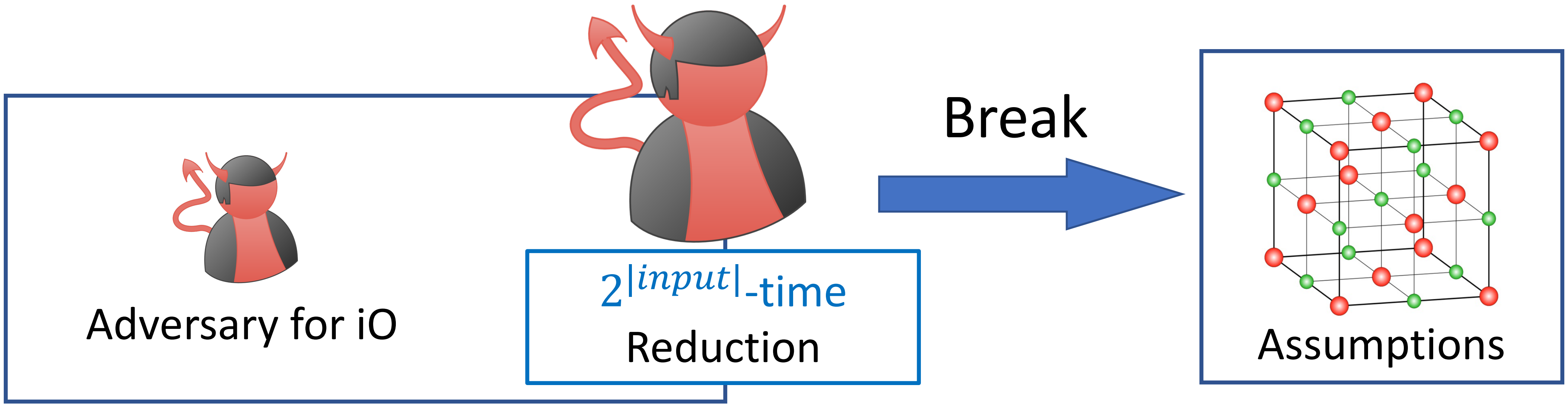
Prior Work: Only for *Bounded*-Input Length

[BGLPT’15][CHJV’15][KLW’15][GS’18]…



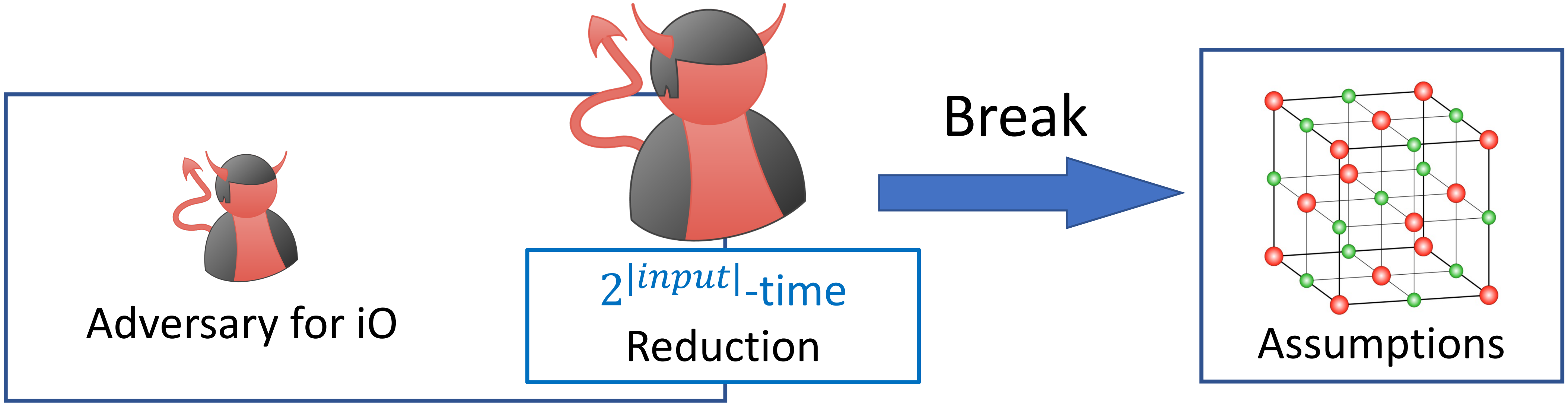
Prior Work: Only for *Bounded*-Input Length

[BGLPT'15][CHJV'15][KLW'15][GS'18]...



Prior Work: Only for *Bounded*-Input Length

[BGLPT'15][CHJV'15][KLW'15][GS'18]...

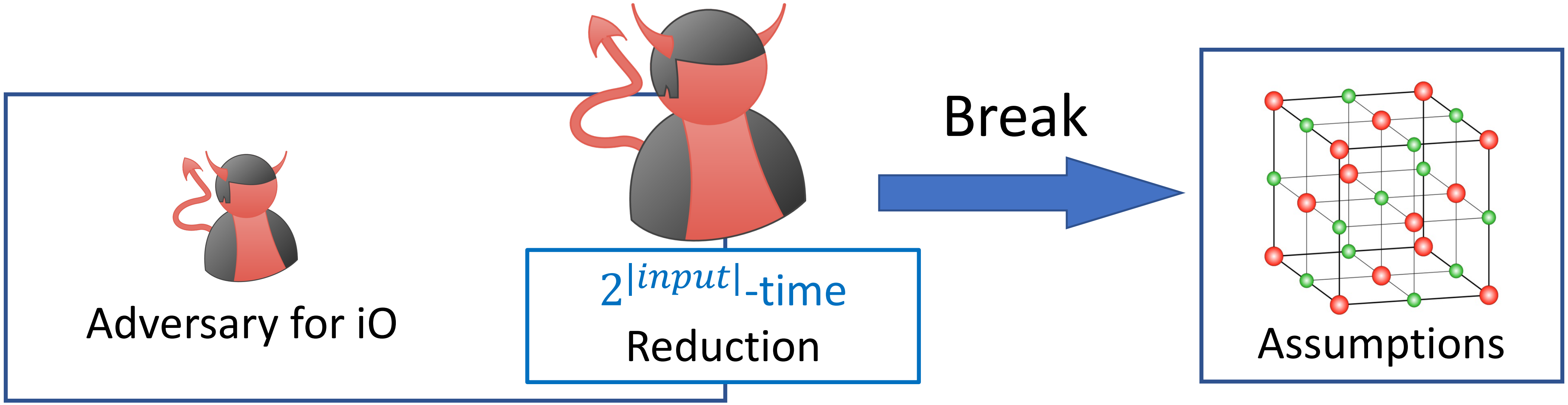


Assume 2^{λ^c}-hardness of assumptions

& set $\boldsymbol{\lambda}$ s.t. $2^{\lambda^c} > 2^{|input|}$

Prior Work: Only for *Bounded*-Input Length

[BGLPT'15][CHJV'15][KLW'15][GS'18]…



Assume 2^{λ^c}-hardness of assumptions
& set $\boldsymbol{\lambda}$ s.t. $2^{\lambda^c} > 2^{|input|}$

$\Rightarrow \; |\boldsymbol{input}| < \boldsymbol{\lambda^c}$

Why $2^{|input|}$ Loss?

Why $2^{|input|}$ Loss?

Real adversary

$C_0 \equiv C_1$

Reduction

Break

Assumptions

Why $2^{|input|}$ Loss?

$C_0 \equiv C_1$

Reduction

Real adversary

Break → Assumptions

$C_0(x^*) \neq C_1(x^*)$

Reduction

Fake adversary

Not Break → Assumptions

Why $2^{|input|}$ Loss?

Reduction

$C_0 \equiv C_1$

Real adversary

Break → Assumptions

Reduction needs to 'decide' the functionality equivalence.

Reduction

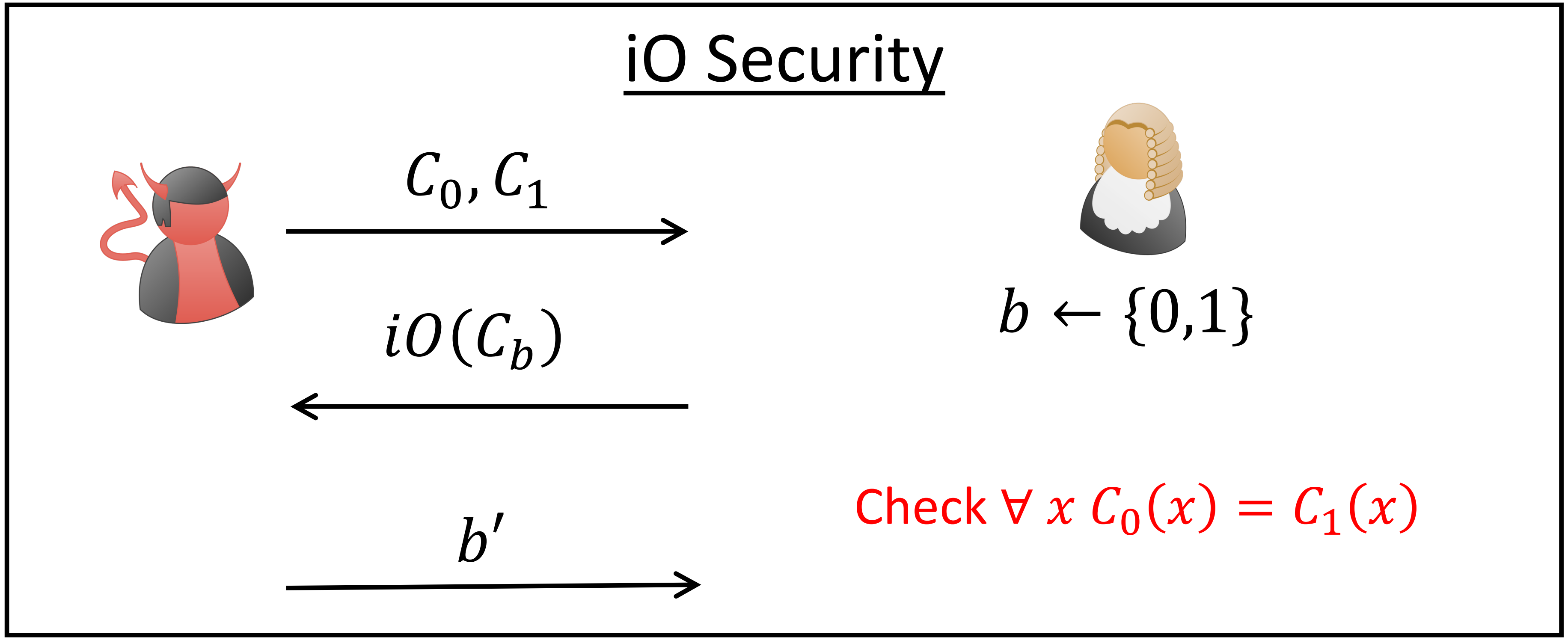
$C_0(x^*) \neq C_1(x^*)$

Fake adversary

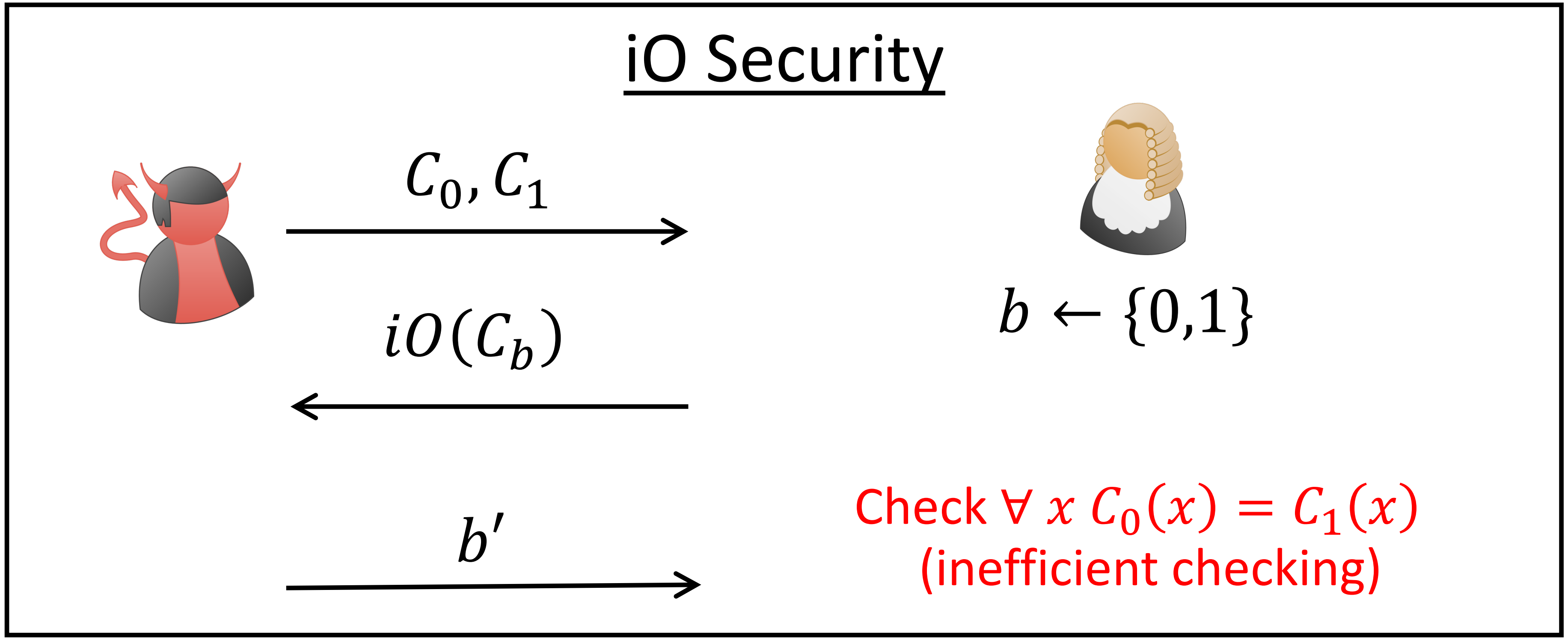
Not Break → Assumptions

Non-Falsifiability

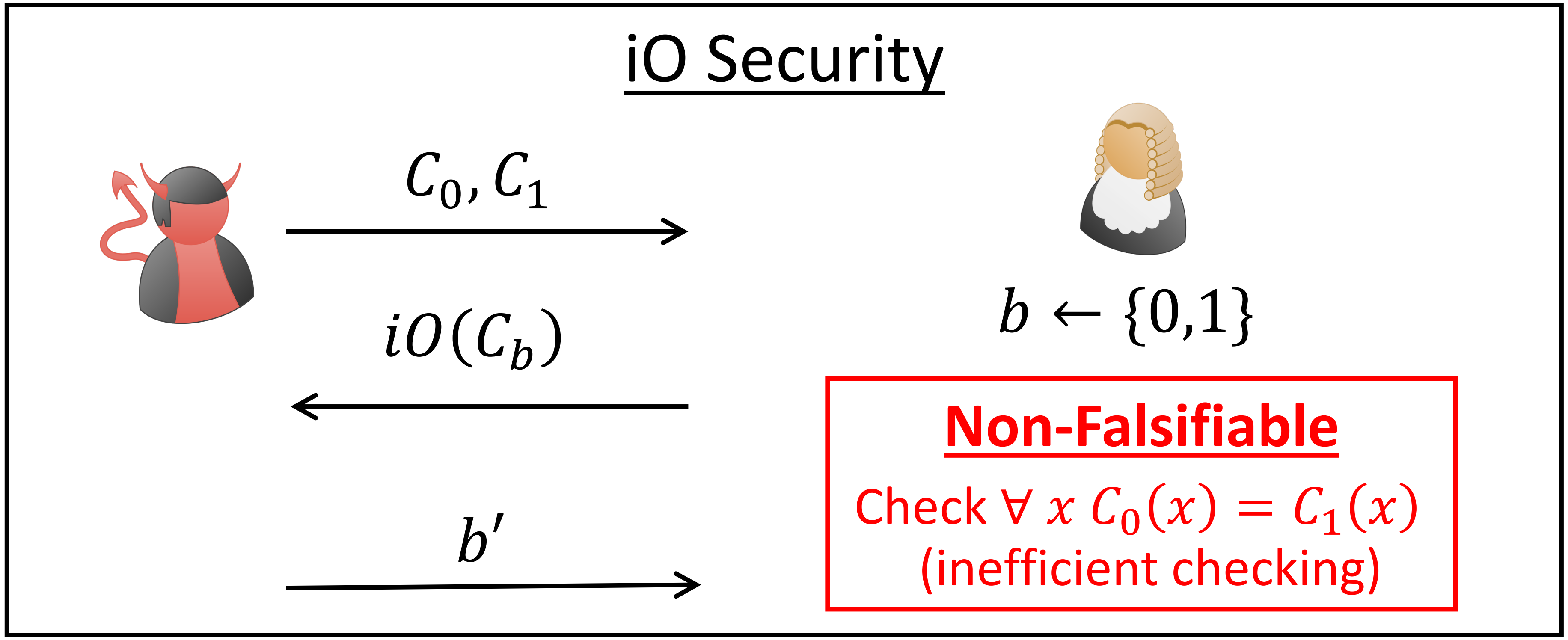
Non-Falsifiability



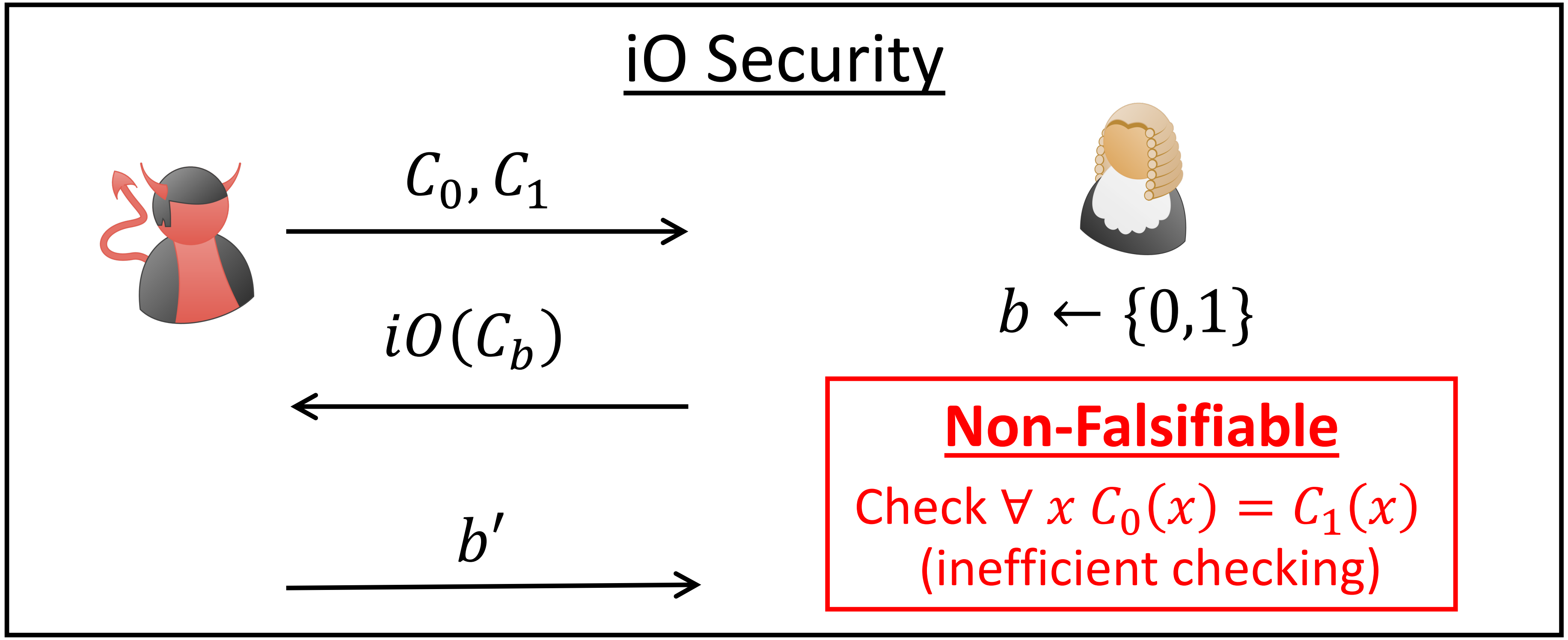
Non-Falsifiability



Non-Falsifiability



Non-Falsifiability



Broader Perspective

Non-Falsifiable definitions appear in many other places, e.g. proof systems. [Gentry-Wichs'10]

This Talk: How to overcome the non-falsifiability barrier?

This Talk: How to overcome the non-falsifiability barrier?

Prior Work

[Garg-Pandey-Srinivasan'16, Garg-Srinivasan'16, Garg-Pandey-Srinivasan-Zhandry'17][Liu-Zhandry'17]:

Require that "$\forall x \; C_0(x) = C_1(x)$" can be decided in **P**

Observation: We *Prove* Equivalence in Math

Applications

iO

Observation: We *Prove* Equivalence in Math

Applications

iO

Security Proof of the Application

Observation: We *Prove* Equivalence in Math

Applications

iO

Security Proof of the Application

Observation: We *Prove* Equivalence in Math

Applications

iO

<u>Security Proof of the Application</u>

Build C_0 , C_1

Observation: We *Prove* Equivalence in Math

Applications

iO

<u>Security Proof of the Application</u>

Build C_0 , C_1

Math proof of $\forall x\ C_0(x) = C_1(x)$

Observation: We *Prove* Equivalence in Math

Applications

iO

Security Proof of the Application

Build C_0 , C_1

Math proof of $\forall x \; C_0(x) = C_1(x)$

iO Security $\Rightarrow iO(C_0) \approx iO(C_1)$

Observation: We *Prove* Equivalence in Math

Applications

iO

Math proof of $\forall x \; C_0(x) = C_1(x)$

Observation: We *Prove* Equivalence in Math

Applications

iO

I can only process *short* proofs.
(length $\ll 2^{|input|}$)

Math proof of $\forall x \; C_0(x) = C_1(x)$

(our little brain)

Our Approach

Short mathematical proof of "$\forall x \; C_0(x) = C_1(x)$"

iO

Our Approach

Short mathematical proof of "$\forall x\ C_0(x) = C_1(x)$"

↓

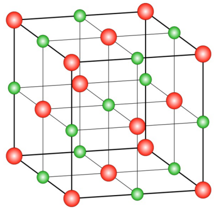
iO

Our Approach

Short mathematical proof of "$\forall x\ C_0(x) = C_1(x)$"

iO

Efficient Reduction Algorithm
(this work)



Assumptions

Our Result

iO for any Turing machines M_1, M_2 with "$\forall x\ M_1(x) = M_2(x)$"

provable in Cook's Theory PV, based on well-founded assumptions.

Cook's Theory PV [Cook'75]

- Polynomial time reasoning

<u>Polynomial-time Induction rule:</u>
"If $\Phi(0)$ is true, and $\forall n, \Phi(n) \rightarrow \Phi(2n) \wedge \Phi(2n+1)$, then $\forall n\, \Phi(n)$."

<u>Can define *any* **polynomial-time functions**</u>, e.g.:

- Arithmetic: $+, -, \times, \div, \leq, <, \lfloor\cdot\rfloor, mod$, ...
- Logic Symbols: $\rightarrow, \neg, \wedge$, ...

What Theorems Can PV Prove?

What Theorems Can PV Prove?

Prior work

What Theorems Can PV Prove?

Prior work

- Correctness of "natural" algorithms in P

What Theorems Can PV Prove?

Prior work

- Correctness of "natural" algorithms in P
- Basic Linear Algebra

What Theorems Can PV Prove?

Prior work

- Correctness of "natural" algorithms in P
- Basic Linear Algebra
- Combinatorial Theorems

What Theorems Can PV Prove?

Prior work

- Correctness of "natural" algorithms in P
- Basic Linear Algebra
- Combinatorial Theorems

…

What Theorems Can PV Prove?

Prior work

- Correctness of "natural" algorithms in P
- Basic Linear Algebra
- Combinatorial Theorems

…

This work

Many crypto algorithms are "natural":

ElGamal Encryption

Regev's Encryption

Puncturable PRFs

…

What Theorems Can PV Prove?

Prior work

- Correctness of "natural" algorithms in P
- Basic Linear Algebra
- Combinatorial Theorems

…

This work

Many crypto algorithms are "natural":

ElGamal Encryption

Regev's Encryption

Puncturable PRFs

...

Unprovable Theorems (assume Factoring is hard)

- Fermat's Little Theorem
- Correctness for "Primes is in P"

How to leverage mathematical proofs?

How to leverage mathematical proofs?

Overview of Techniques

What Information does a Proof Provide?

What Information does a Proof Provide?

Mathematical Proofs Have Structures

1.	$P \wedge Q$	Premise
2.	P	Decomposing a conjunction (1)
3.	Q	Decomposing a conjunction (1)
4.	$P \rightarrow \neg(Q \wedge R)$	Premise
5.	$\neg(Q \wedge R)$	Modus ponens (3,4)
6.	$\neg Q \vee \neg R$	DeMorgan (5)
7.	$\neg R$	Disjunctive syllogism (3,6)
8.	$S \rightarrow R$	Premise
9.	$\neg S$	Modus tollens (7,8) $\square$

What Information does a Proof Provide?

Mathematical Proofs Have Structures

1.	$P \wedge Q$	Premise
2.	P	Decomposing a conjunction (1)
3.	Q	Decomposing a conjunction (1)
4.	$P \to \neg(Q \wedge R)$	Premise
5.	$\neg(Q \wedge R)$	Modus ponens (3,4)
6.	$\neg Q \vee \neg R$	DeMorgan (5)
7.	$\neg R$	Disjunctive syllogism (3,6)
8.	$S \to R$	Premise
9.	$\neg S$	Modus tollens (7,8) $\square$

- **Localness:** Each line is derived from O(1) previous lines

What Information does a Proof Provide?

Mathematical Proofs Have Structures

1.	$P \wedge Q$	Premise
2.	P	Decomposing a conjunction (1)
3.	Q	Decomposing a conjunction (1)
4.	$P \to \neg(Q \wedge R)$	Premise
5.	$\neg(Q \wedge R)$	Modus ponens (3,4)
6.	$\neg Q \vee \neg R$	DeMorgan (5)
7.	$\neg R$	Disjunctive syllogism (3,6)
8.	$S \to R$	Premise
9.	$\neg S$	Modus tollens (7,8) □

- **Localness:** Each line is derived from O(1) previous lines
- In Propositional Logic (Extended Frege): each line is also a *circuit*

What Information does a Proof Provide?

Mathematical Proofs Have Structures

1.	$P \wedge Q$	Premise
2.	P	Decomposing a conjunction (1)
3.	Q	Decomposing a conjunction (1)
4.	$P \to \neg(Q \wedge R)$	Premise
5.	$\neg(Q \wedge R)$	Modus ponens (3,4)
6.	$\neg Q \vee \neg R$	DeMorgan (5)
7.	$\neg R$	Disjunctive syllogism (3,6)
8.	$S \to R$	Premise
9.	$\neg S$	Modus tollens (7,8) $\square$

- **Localness:** Each line is derived from O(1) previous lines
- In Propositional Logic (Extended Frege): each line is also a *circuit*

Rest of the Talk: mainly focus on extended Frege ($\mathcal{EF}$), since PV-proof can be translated to $\mathcal{EF}$-Proof.

Bypass $2^{|\text{input}|}$-Loss via $\mathcal{EF}$-Proofs

Bypass $2^{|\text{input}|}$-Loss via $\mathcal{EF}$-Proofs

Hybrid Argument

$iO(C_0)$ $iO(C_1)$

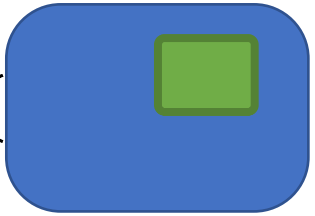
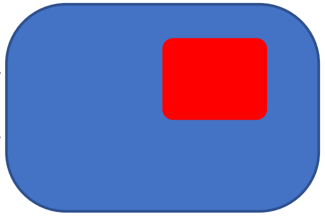
Bypass $2^{|\text{input}|}$-Loss via $\mathcal{EF}$-Proofs

Hybrid Argument

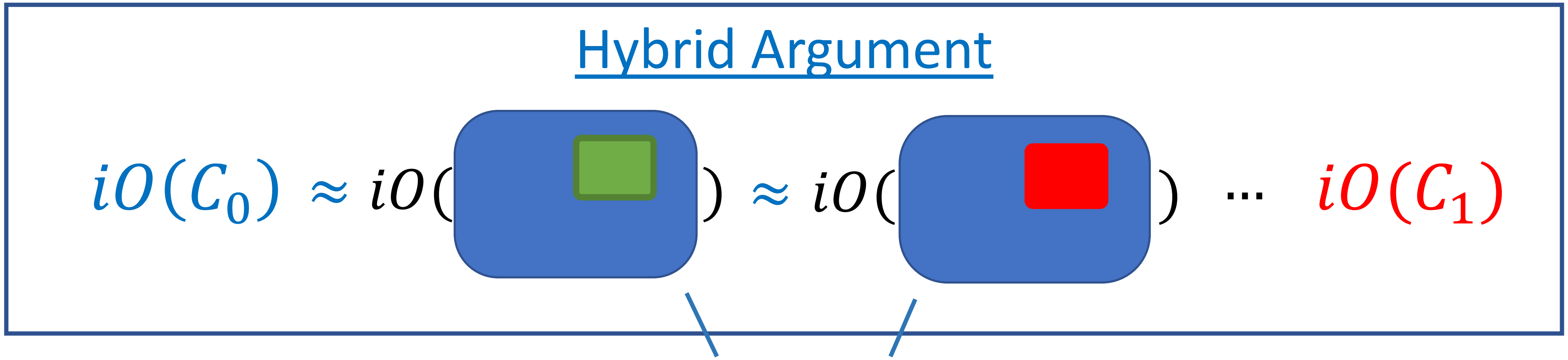
$$iO(C_0) \approx iO(\qquad) \approx iO(\qquad) \cdots \quad iO(C_1)$$

Bypass $2^{|\text{input}|}$-Loss via $\mathcal{EF}$-Proofs

Hybrid Argument

$iO(C_0) \approx iO($  $) \approx iO($  $)$ ⋯ $iO(C_1)$

Bypass $2^{|\text{input}|}$-Loss via $\mathcal{EF}$-Proofs

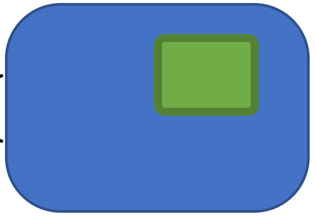
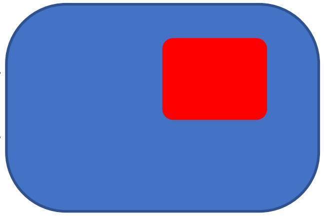


"Locally equivalent", checkable *in polynomial time*

Bypass $2^{|\text{input}|}$-Loss via $\mathcal{EF}$-Proofs

We build iO for locally equivalent circuits
with loss independent of |input|.

<u>Hybrid Argument</u>

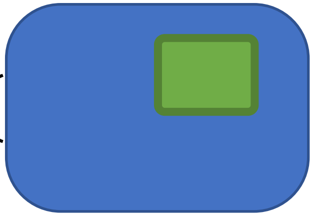
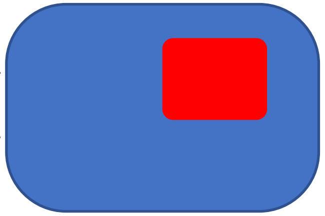
$iO(C_0) \approx iO($  $) \approx iO($  $)$ … $iO(C_1)$

“***Locally*** equivalent”, checkable *in polynomial time*

Bypass $2^{|\text{input}|}$-Loss via $\mathcal{EF}$-Proofs

We build iO for locally equivalent circuits
with loss independent of |input|.

<u>Hybrid Argument</u>

$iO(C_0) \approx iO($  $) \approx iO($  $)$ … $iO(C_1)$

*"**Locally** equivalent"*, checkable *in polynomial time*

Poly. size $\mathcal{EF}$-proof for $C_0(x) \equiv C_1(x)$

Technical Details

- $\mathcal{EF}$-Proofs ⇒ local equivalence
- iO for locally equivalent ckts
- iO for Turing machines

Technical Details

- **$\mathcal{EF}$-Proofs ⇒ local equivalence**
- iO for locally equivalent ckts
- iO for Turing machines

Define Local Equivalence

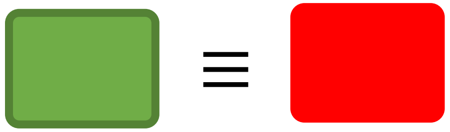
Define Local Equivalence

Define Local Equivalence

C:

: C'

C and C' are almost the same (with same topology), except for a **functionality equivalent *__sub-circuit__*** of size $O(\log n)$

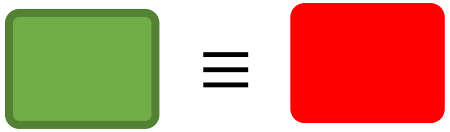


Define Local Equivalence

C:

$: C'$

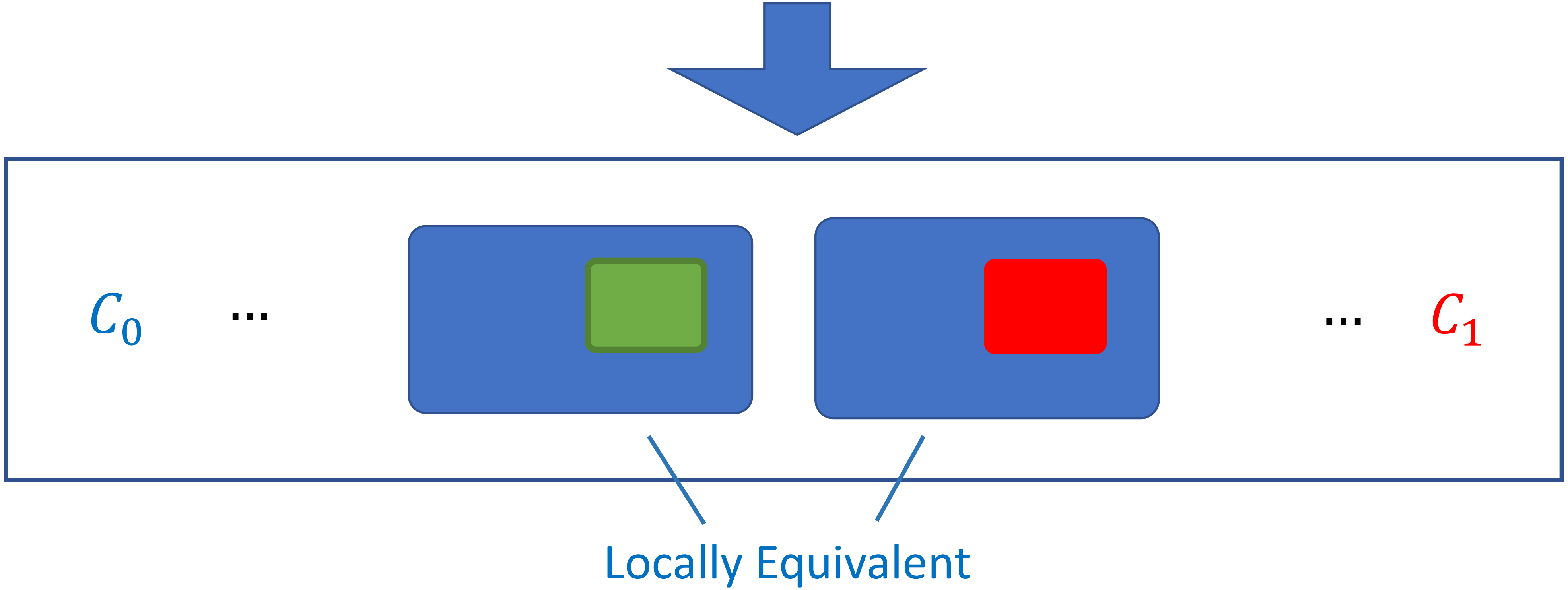
C and $C'$are almost the same (with same topology), except for a **functionality equivalent *<u>sub-circuit</u>*** of size $O(\log n)$



(Sub-circuit: induced subgraph from a subset of gates)

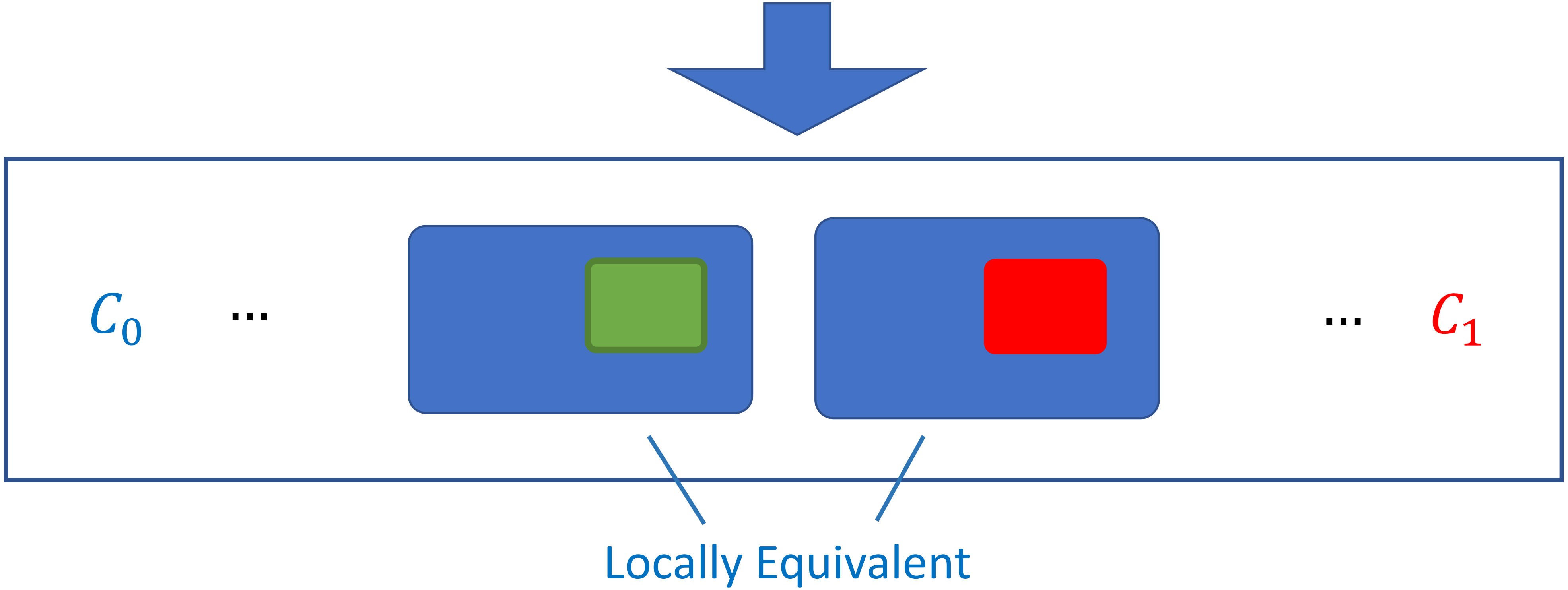
Goal: $\mathcal{EF}$-Proof ⇒ Locally Equivalent Circuits

$\mathcal{EF}$ proof for $C_0(x) \equiv C_1(x)$



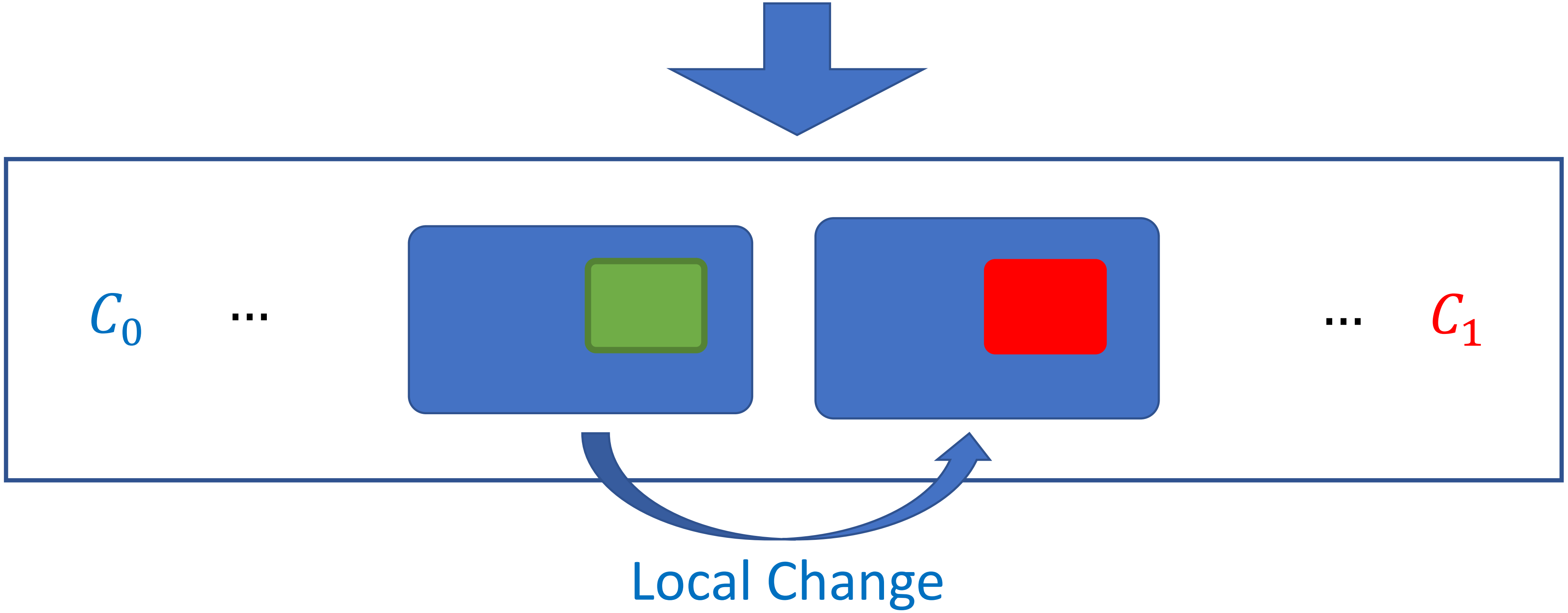
Alternative View: A Series of Local Changes

$\mathcal{EF}$ proof for $C_0(x) \equiv C_1(x)$



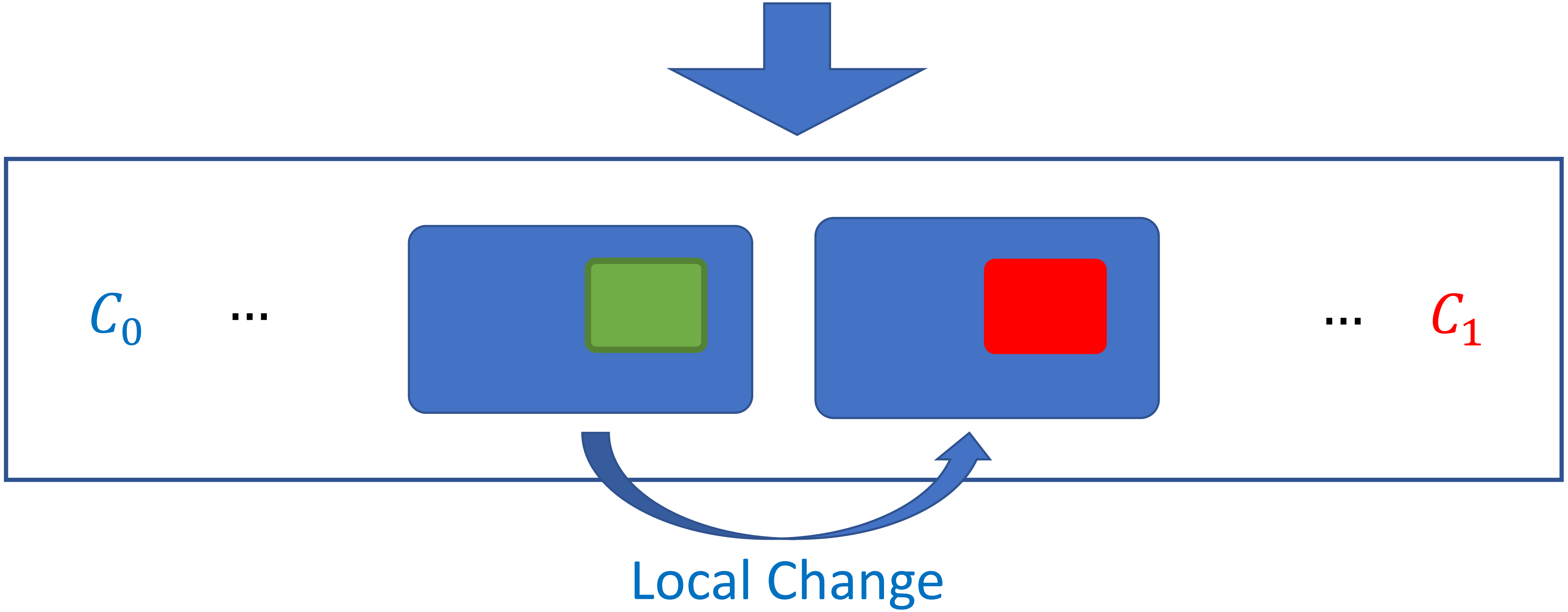
Alternative View: A Series of Local Changes

$\mathcal{EF}$ proof for $C_0(x) \equiv C_1(x)$



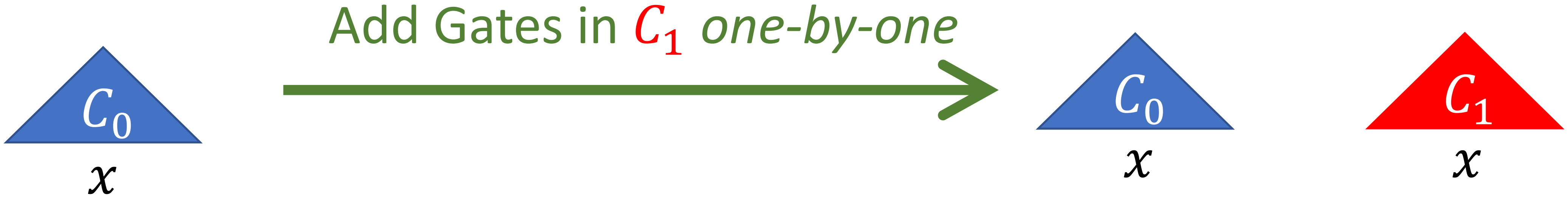
Alternative View: A Series of Local Changes

$\mathcal{EF}$ proof for $C_0(x) \equiv C_1(x)$

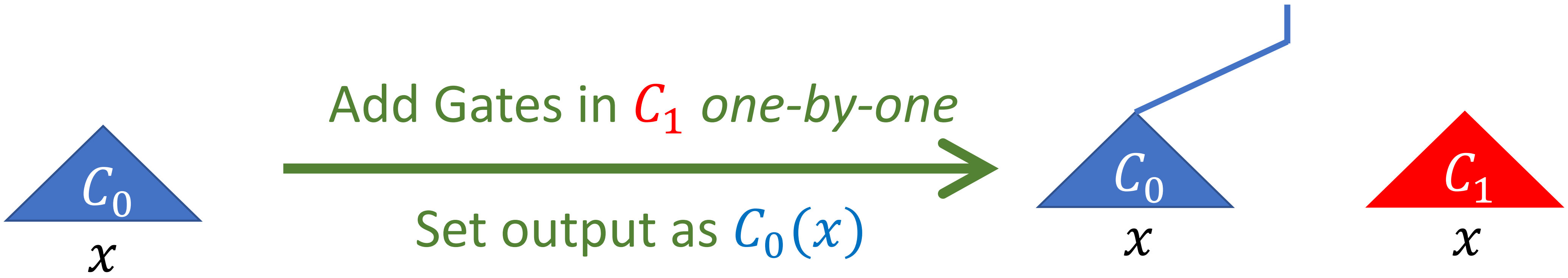


Simplification in This Talk: Ignore topology & allow multi-arity gates

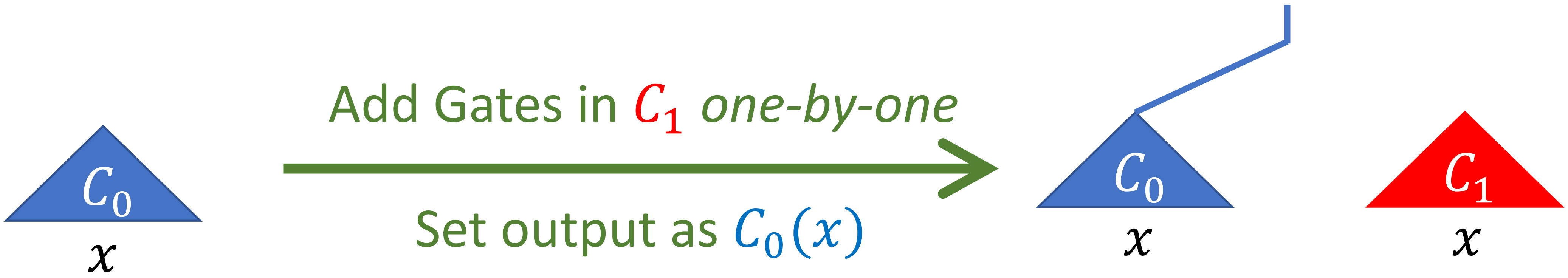
Stage I: Grow C_1



Stage I: Grow C_1



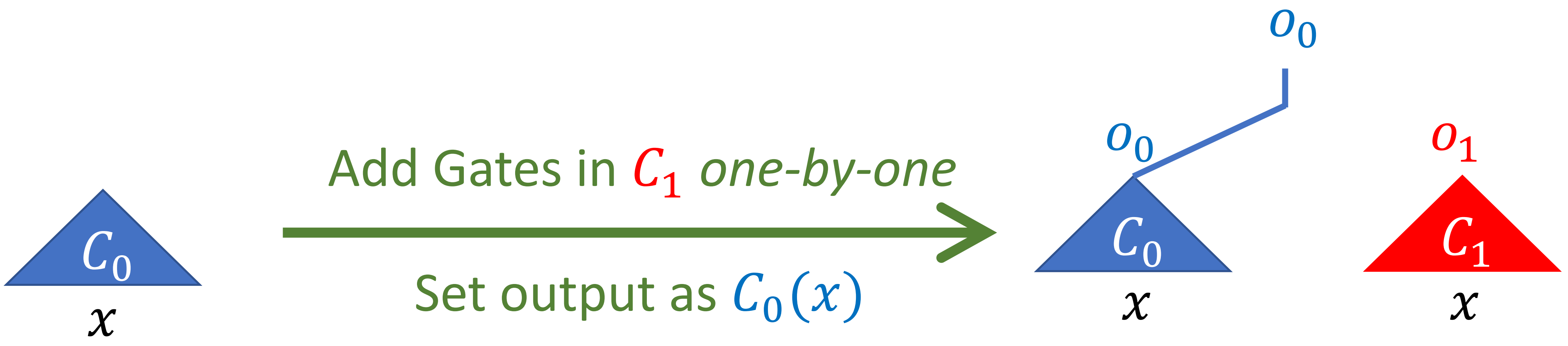
Stage I: Grow C_1



Local Equivalence

When a gate is added, its output is not used anywhere

Stage I: Grow C_1



Local Equivalence

When a gate is added, its output is not used anywhere

Stage II: Grow the Proof

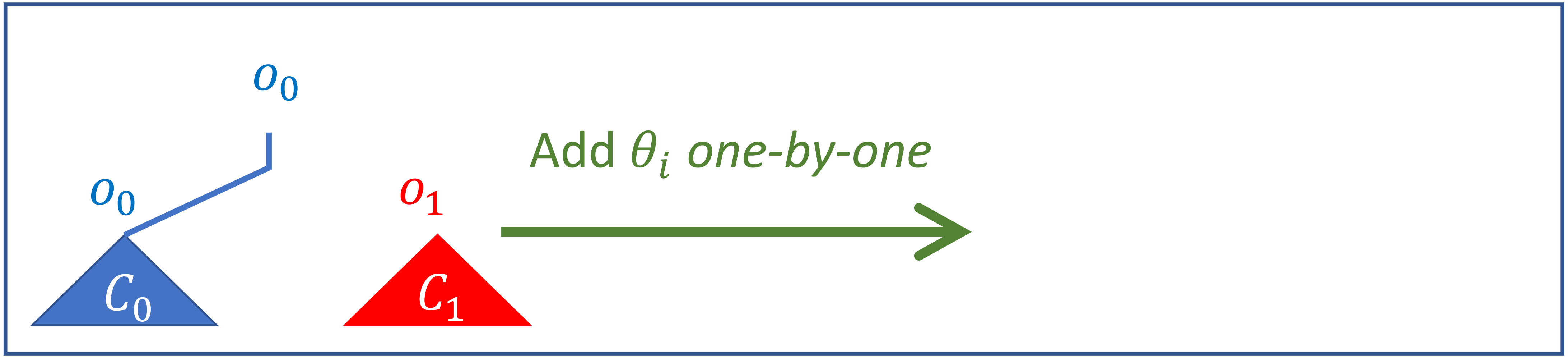
$\mathcal{EF}$-Proof of $C_0(x) \leftrightarrow C_1(x)$: $\theta_1, \theta_2, \ldots, \theta_\ell$

Stage II: Grow the Proof

$\mathcal{EF}$-Proof of $C_0(x) \leftrightarrow C_1(x)$: $\theta_1, \theta_2, \ldots, \theta_\ell$

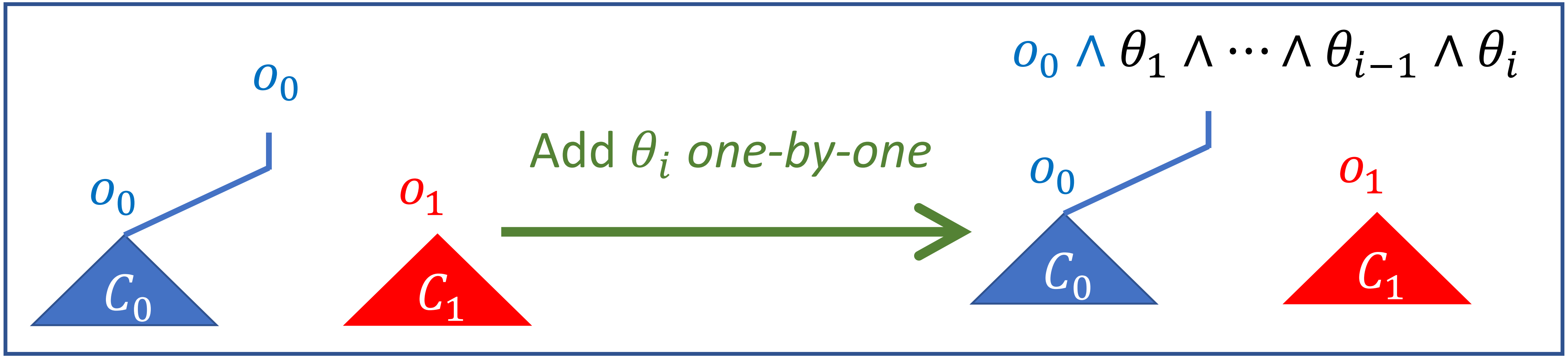
Stage II: Grow the Proof

$\mathcal{EF}$-Proof of $C_0(x) \leftrightarrow C_1(x)$: $\theta_1, \theta_2, \ldots, \theta_\ell$



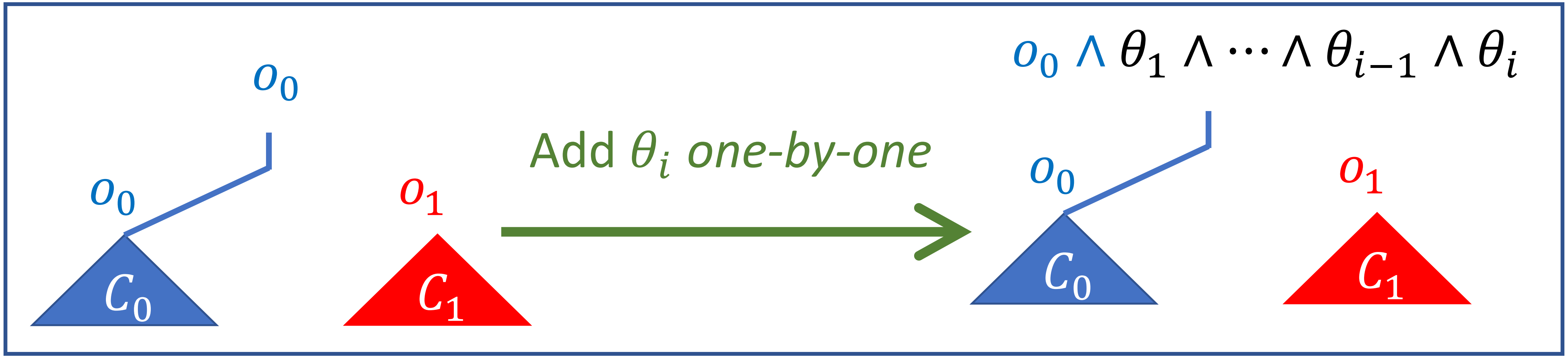
Stage II: Grow the Proof

$\mathcal{EF}$-Proof of $C_0(x) \leftrightarrow C_1(x)$: $\theta_1, \theta_2, \ldots, \theta_\ell$



Stage II: Grow the Proof

$\mathcal{EF}$-Proof of $C_0(x) \leftrightarrow C_1(x)$: $\theta_1, \theta_2, \dots, \theta_\ell$



Intuition: θ_i's (i.e. lines of the proof) are "true", so the functionality is preserved.

Stage II: Local Equivalence

<u>i-th Step: Add θ_i</u>

<u>Before:</u> $C_0(x) \wedge \theta_1 \wedge \cdots \wedge \theta_{i-1}$

<u>After:</u> $C_0(x) \wedge \theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_i$

Stage II: Local Equivalence

<u>i-th Step: Add θ_i</u>

Before: $C_0(x) \wedge \theta_1 \wedge \cdots \wedge \theta_{i-1}$

After: $C_0(x) \wedge \theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_i$

<u>θ_i is derived via Modus Ponens:</u> $p, p \rightarrow q \vdash q$

Stage II: Local Equivalence

i-th Step: Add θ_i

Before:

After:

θ_i is derived via Modus Ponens: $p, p \rightarrow q \vdash q$

Stage II: Local Equivalence

<u>i-th Step: Add θ_i</u>

Before: $C_0(x) \land p \land \cdots \land (p \to q) \land \cdots$

After: $C_0(x) \land p \land \cdots \land (p \to q) \land \cdots \land q$

<u>θ_i is derived via Modus Ponens:</u> $p, p \to q \vdash q$

Stage II: Local Equivalence

<u>i-th Step: Add θ_i</u>

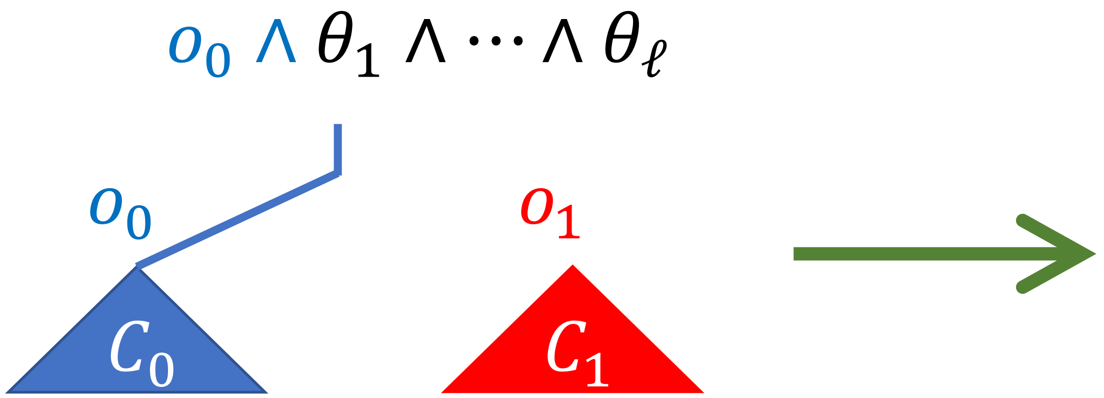
<u>Before:</u> $C_0(x) \wedge p \wedge \cdots \wedge (p \rightarrow q) \wedge \cdots$

<u>After:</u> $C_0(x) \wedge p \wedge \cdots \wedge (p \rightarrow q) \wedge \cdots \wedge q$

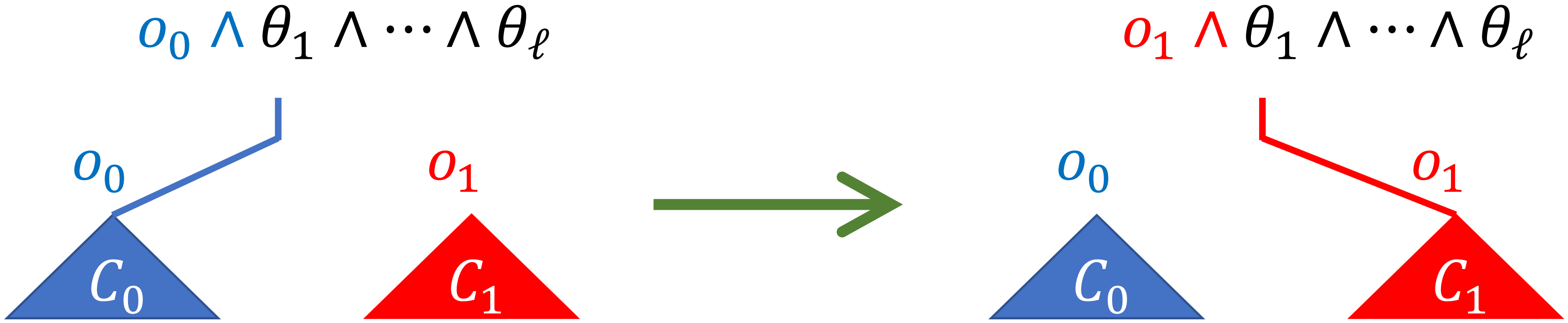
<u>θ_i is derived via Modus Ponens:</u> $p, p \rightarrow q \vdash q$

$$p \wedge (p \rightarrow q) \equiv p \wedge (p \rightarrow q) \wedge q$$

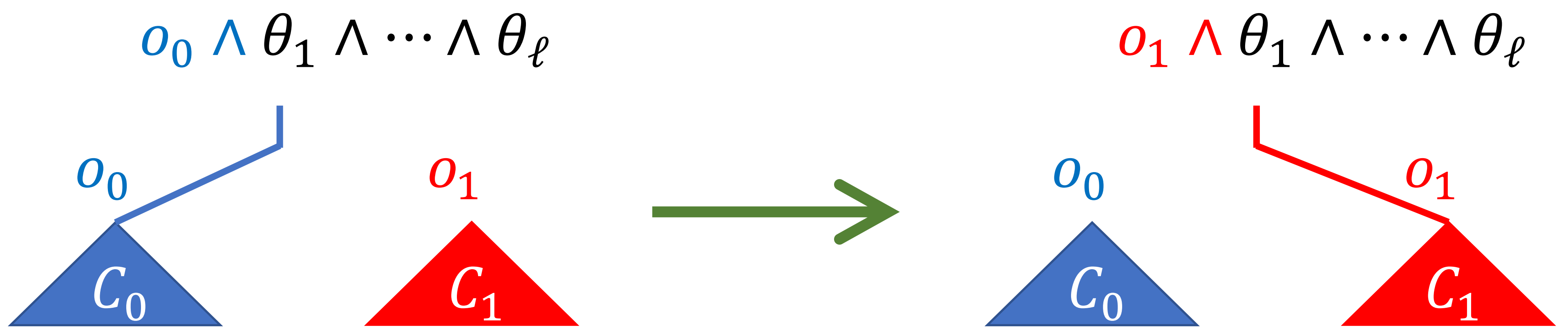
Stage III: **Switch** o_0 to o_1



Stage III: **Switch** o_0 to o_1



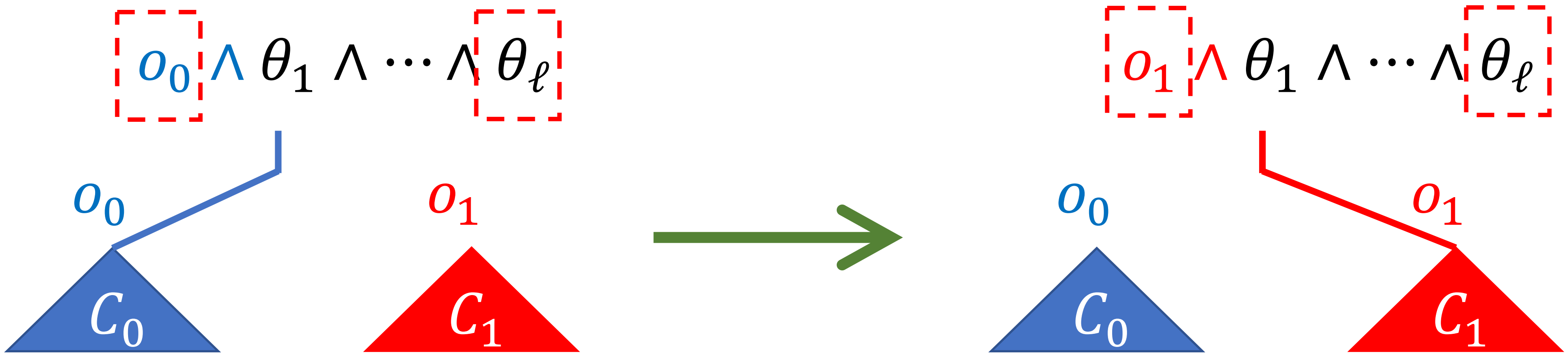
Stage III: **Switch** o_0 to o_1



Local Equivalence

θ_ℓ is "$o_0 \leftrightarrow o_1$" (A proof of $C_0(x) \leftrightarrow C_1(x)$ must end with $o_0 \leftrightarrow o_1$)

Stage III: **Switch** o_0 to o_1



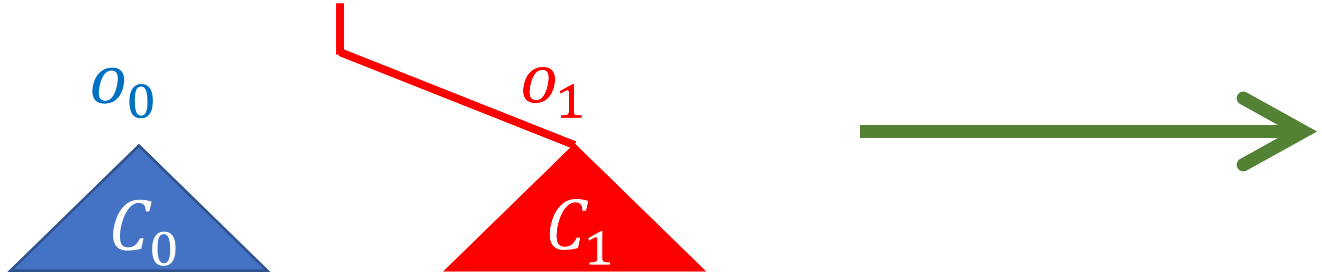
Local Equivalence

θ_ℓ is "$o_0 \leftrightarrow o_1$" (A proof of $C_0(x) \leftrightarrow C_1(x)$ must end with $o_0 \leftrightarrow o_1$)

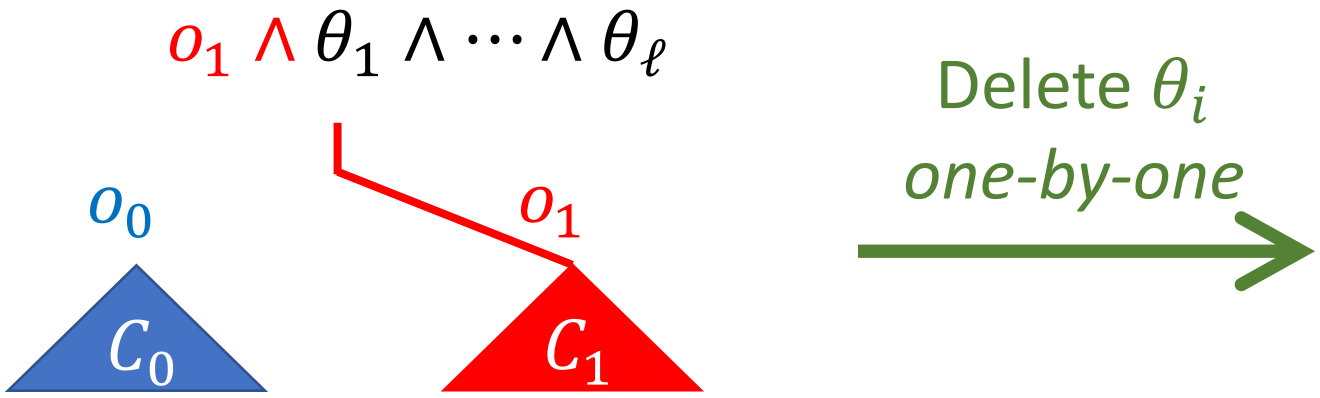
$$o_0 \wedge (o_0 \leftrightarrow o_1) \equiv o_1 \wedge (o_0 \leftrightarrow o_1)$$

Stage IV: Shrink the Proof

$o_1 \wedge \theta_1 \wedge \cdots \wedge \theta_\ell$



Stage IV: Shrink the Proof



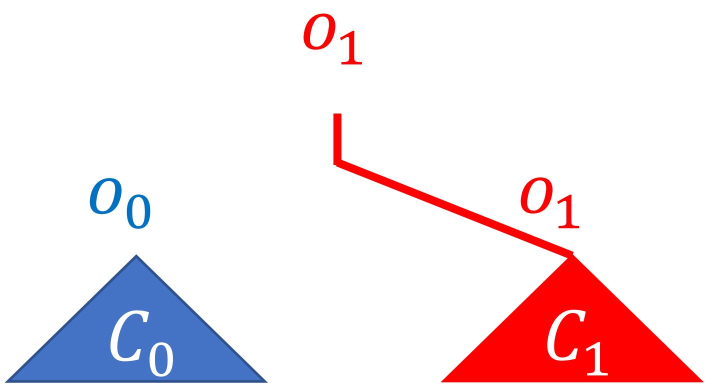
Stage IV: Shrink the Proof

Stage IV: Shrink the Proof

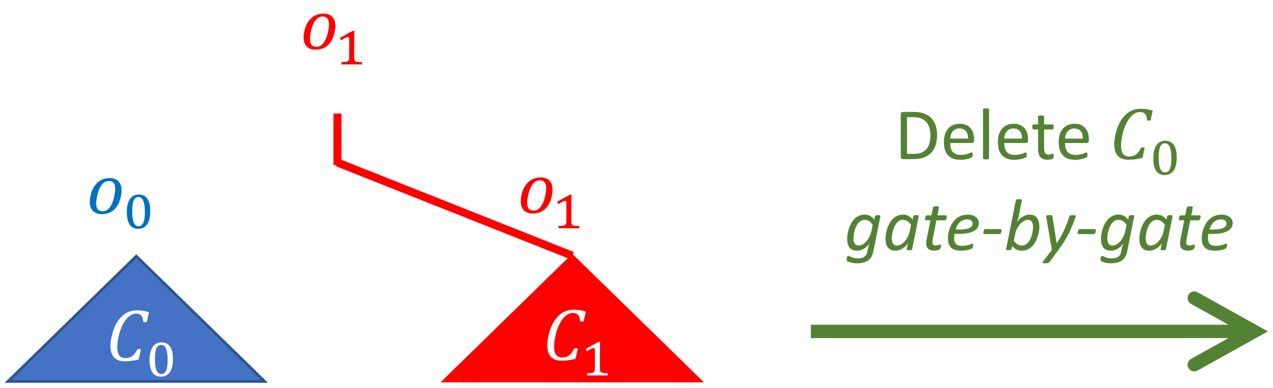
Local Equivalence: Similar to "Grow the proof" Stage

Stage V: Shrink C_0

Stage V: Shrink C_0



Stage V: Shrink C_0



Stage V: Shrink C_0

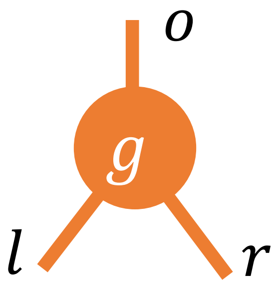
Stage V: Shrink C_0

Local Equivalence: Similar to "Grow C_1" Stage

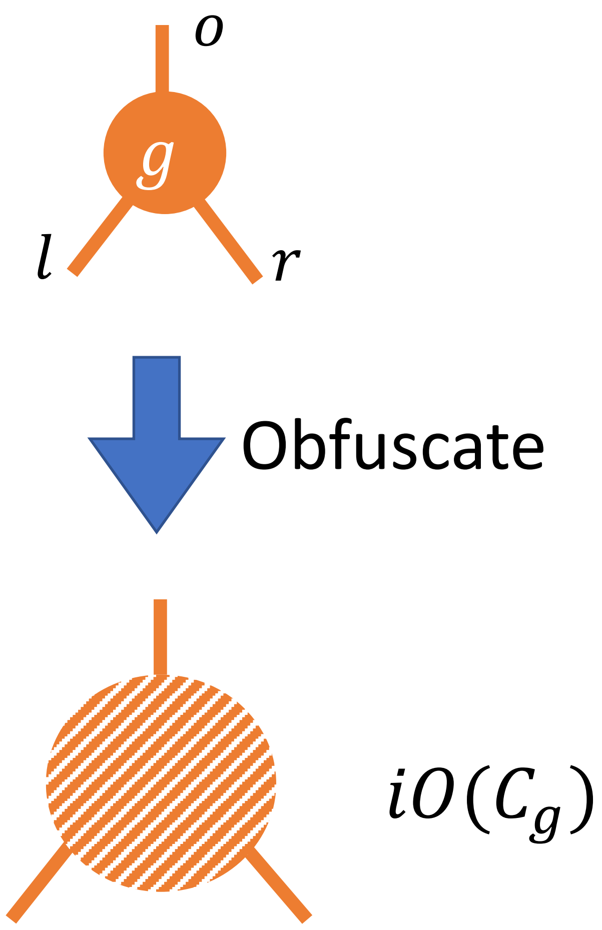
Technical Details

- $\mathcal{EF}$-Proofs $\Rightarrow$ local equivalence
- **iO for locally equivalent ckts**
- iO for Turing machines

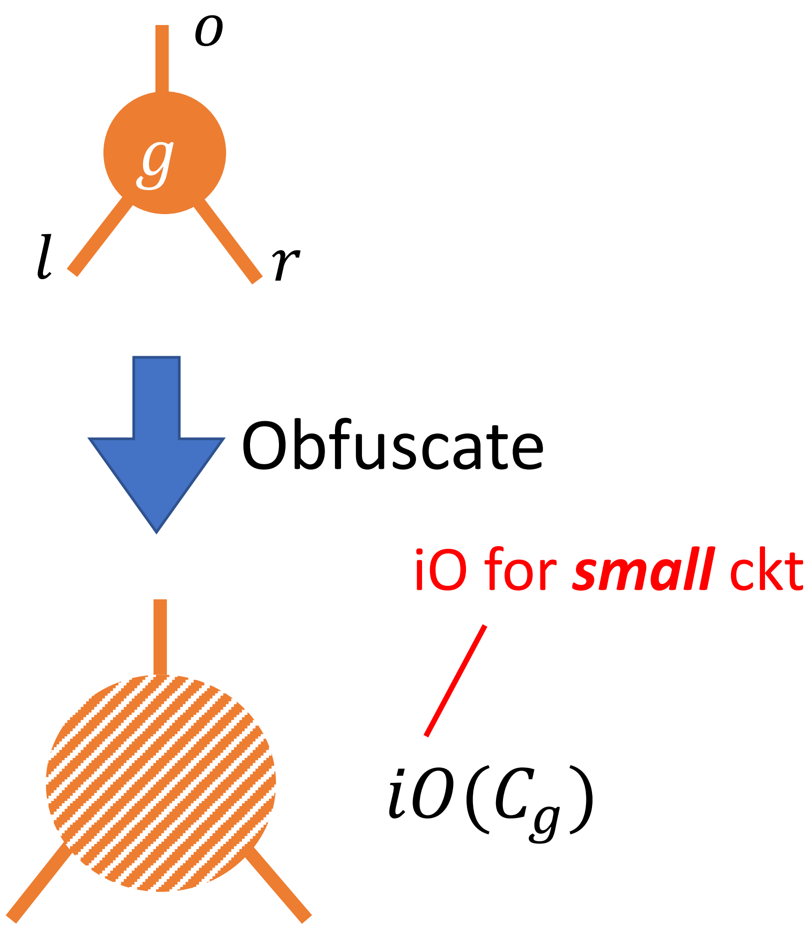
Gate-by-Gate Obfuscation



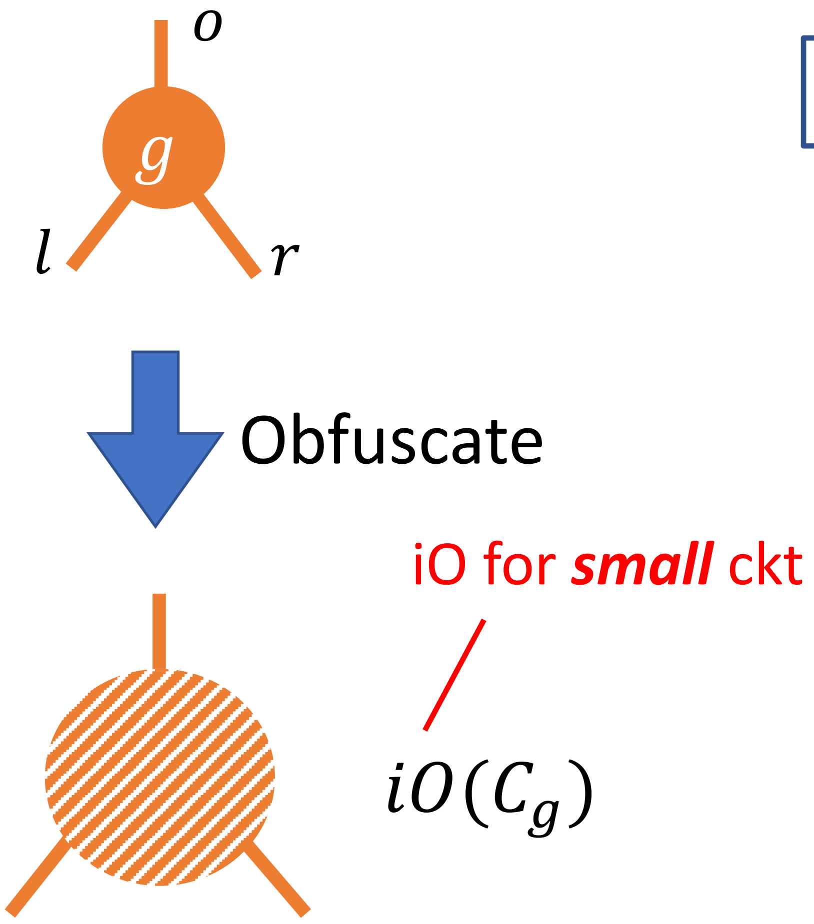
Gate-by-Gate Obfuscation



Gate-by-Gate Obfuscation

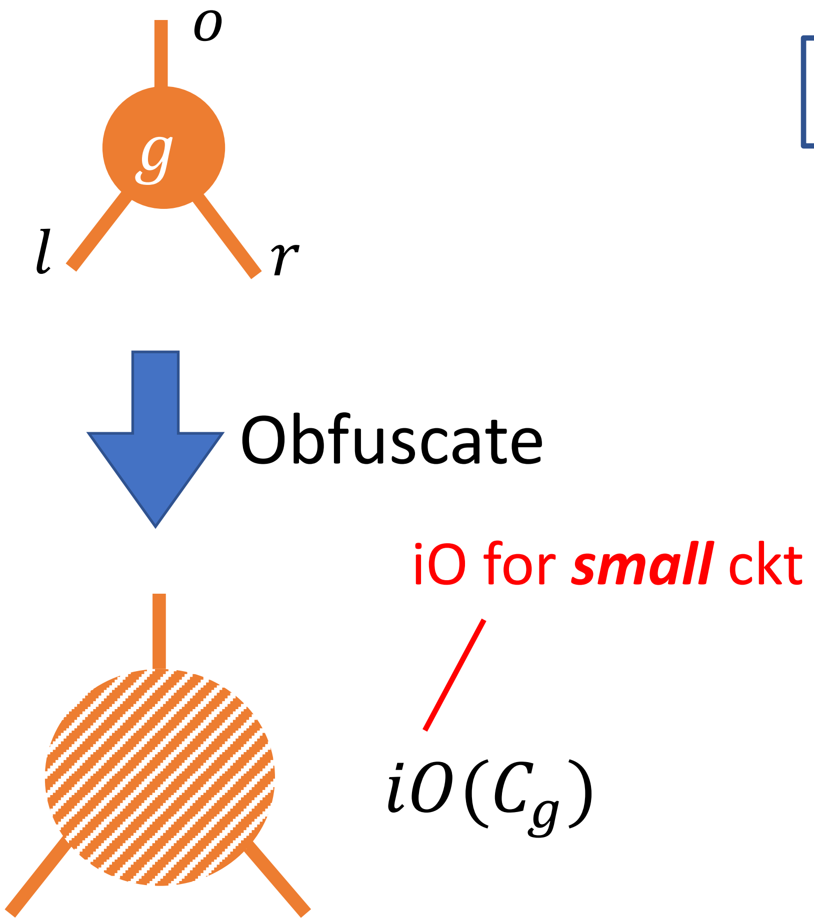


Gate-by-Gate Obfuscation



: encrypt and sign the wire values

Gate-by-Gate Obfuscation



□ : encrypt and sign the wire values

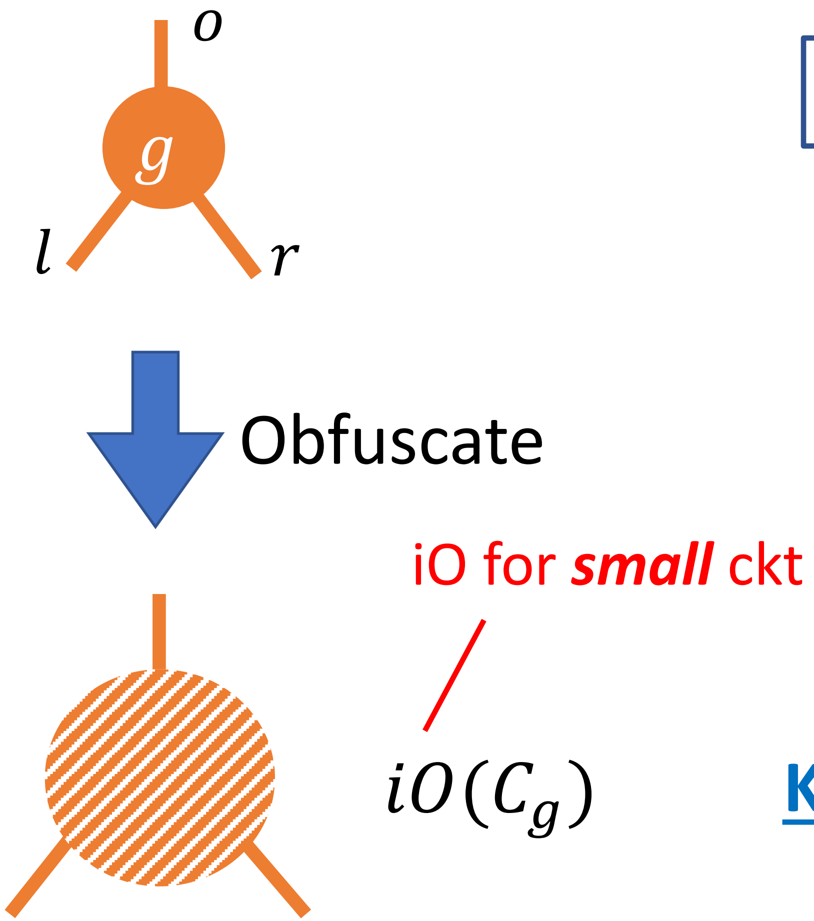
$C_g($ $\boxed{m_l}$ $\boxed{m_r}$ $)$

Decrypt m_l, m_r

$m_o = g(w_l, w_r)$

Output: $\boxed{m_o}$

Gate-by-Gate Obfuscation



☐ : encrypt and sign the wire values

$$C_g(\boxed{m_l}\ \boxed{m_r})$$

Decrypt m_l, m_r

$m_o = g(w_l, w_r)$

Output: $\boxed{m_o}$

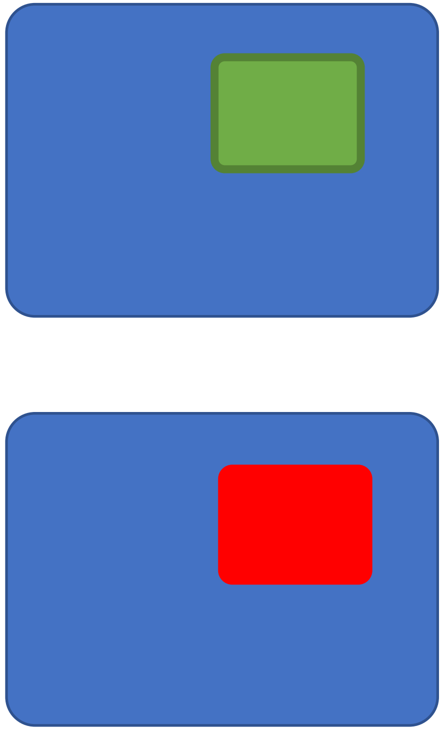
Key Feature: obfuscated circuit preserves the ***topology*** of the input circuit

Prove Security w/o $2^{|\mathrm{input}|}$ Loss

Prove Security w/o $2^{|\mathrm{input}|}$ Loss

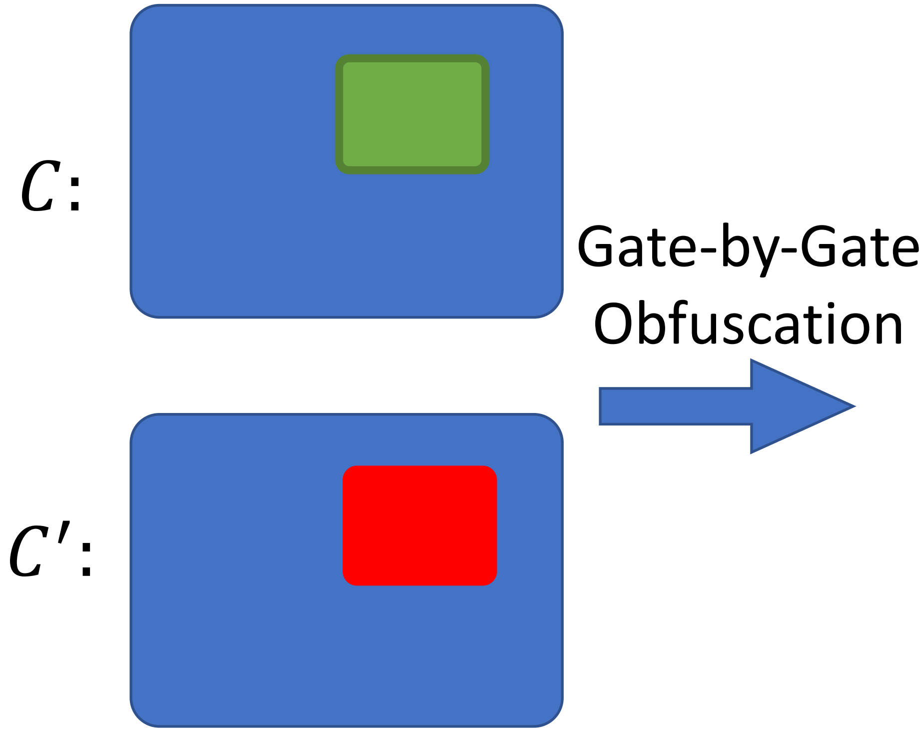
C:

C':



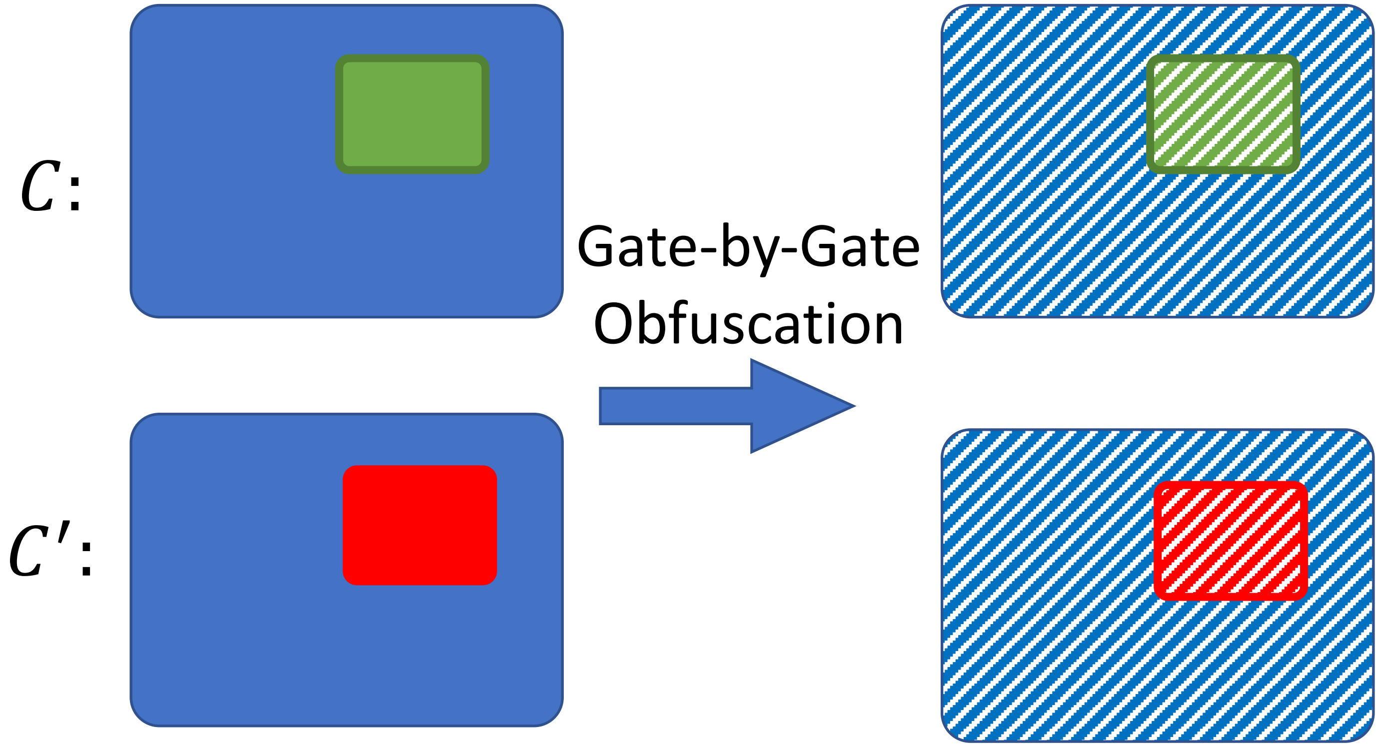
C, C': Locally Equivalent

Prove Security w/o $2^{|\mathrm{input}|}$ Loss



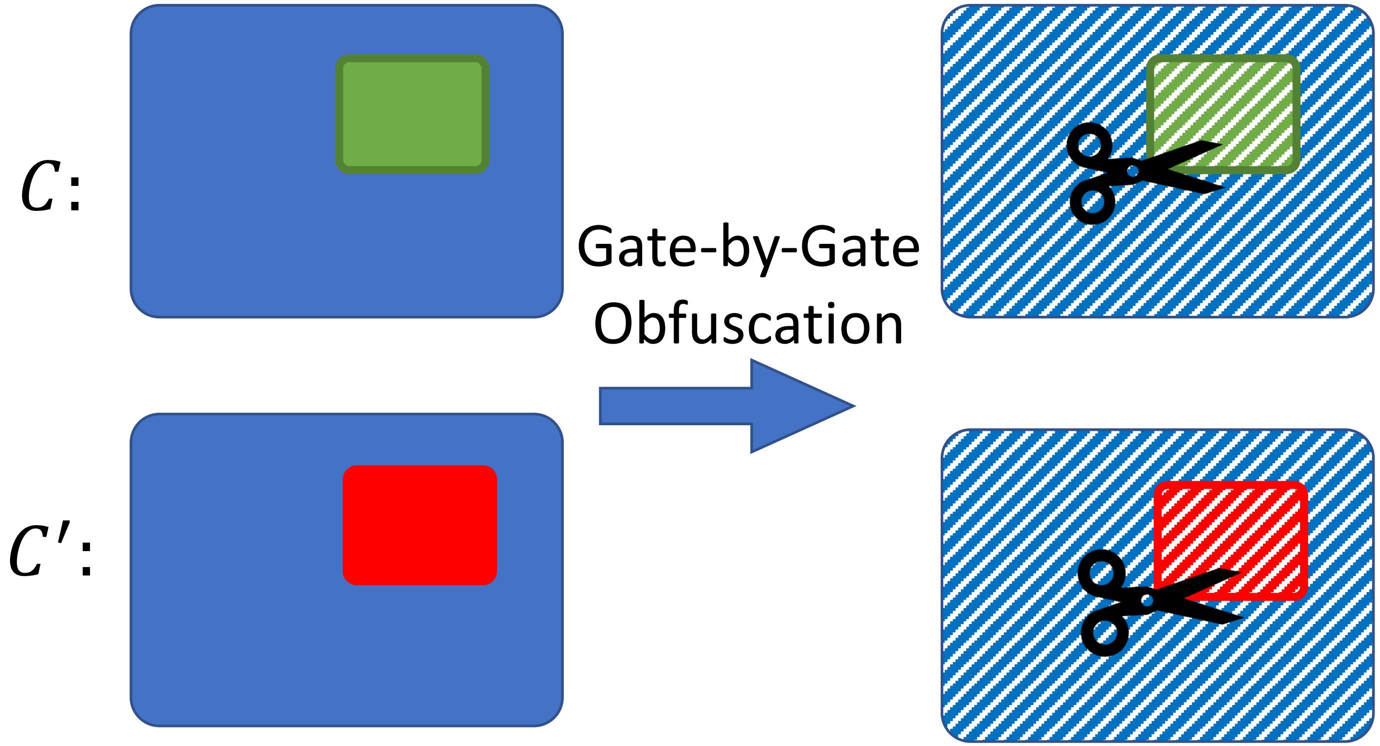
C, C': Locally Equivalent

Prove Security w/o $2^{|\mathrm{input}|}$ Loss



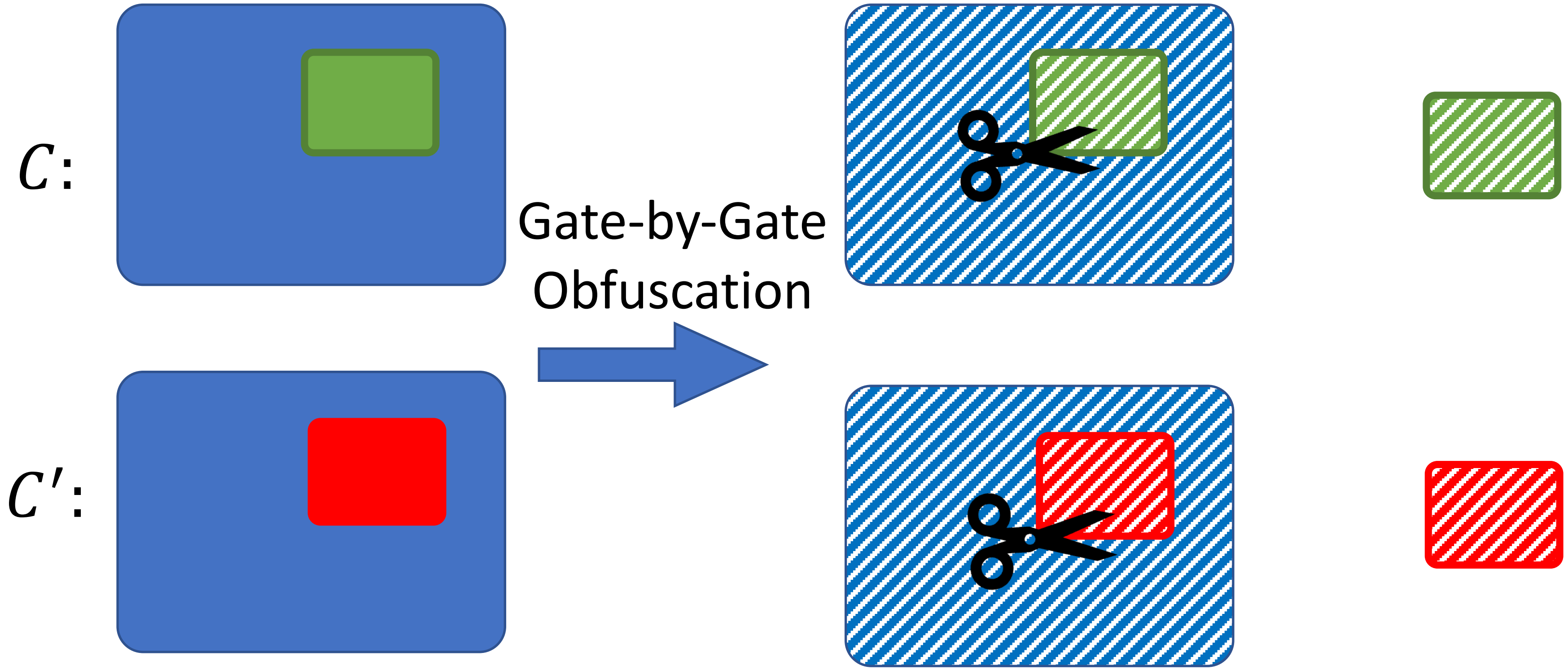
C, C': Locally Equivalent

Prove Security w/o $2^{|\text{input}|}$ Loss



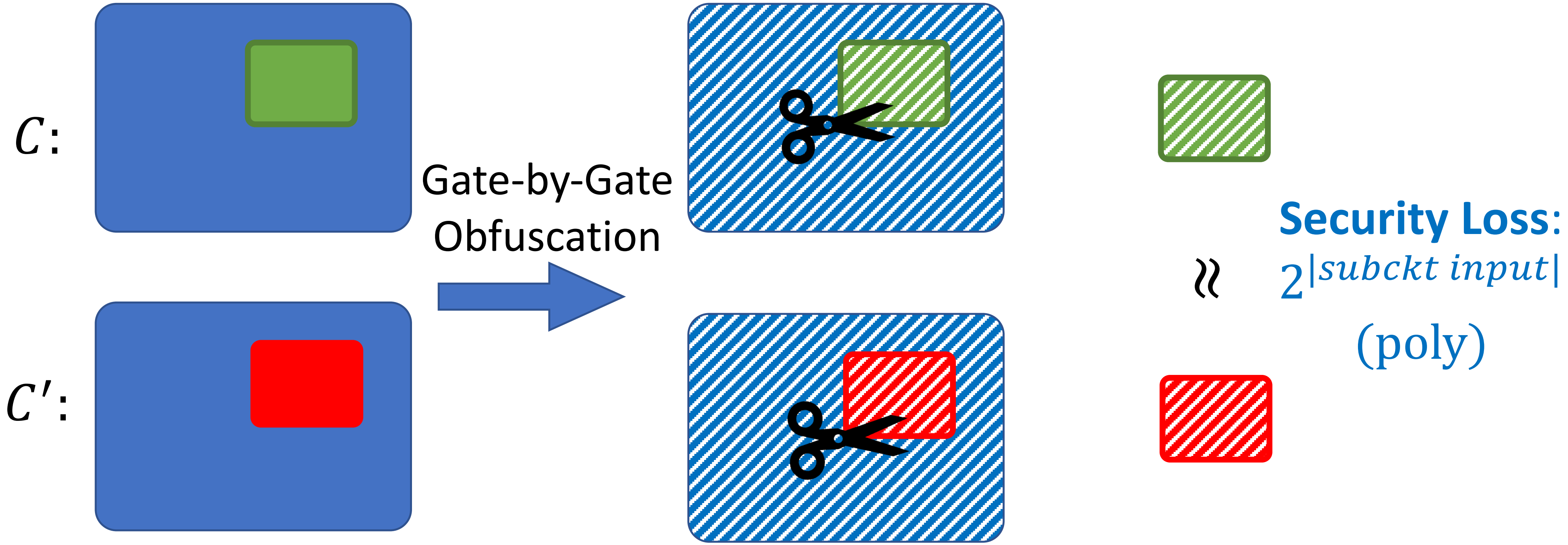
C, C': Locally Equivalent

Prove Security w/o $2^{|\mathrm{input}|}$ Loss



C, C': Locally Equivalent

Prove Security w/o $2^{|\mathrm{input}|}$ Loss

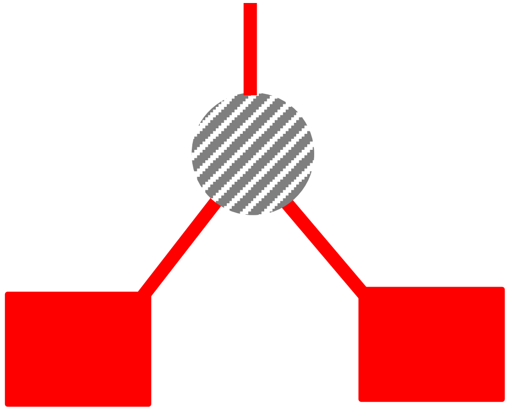


C, C': Locally Equivalent

Challenge: Mix-and-Match Attack

Challenge: Mix-and-Match Attack

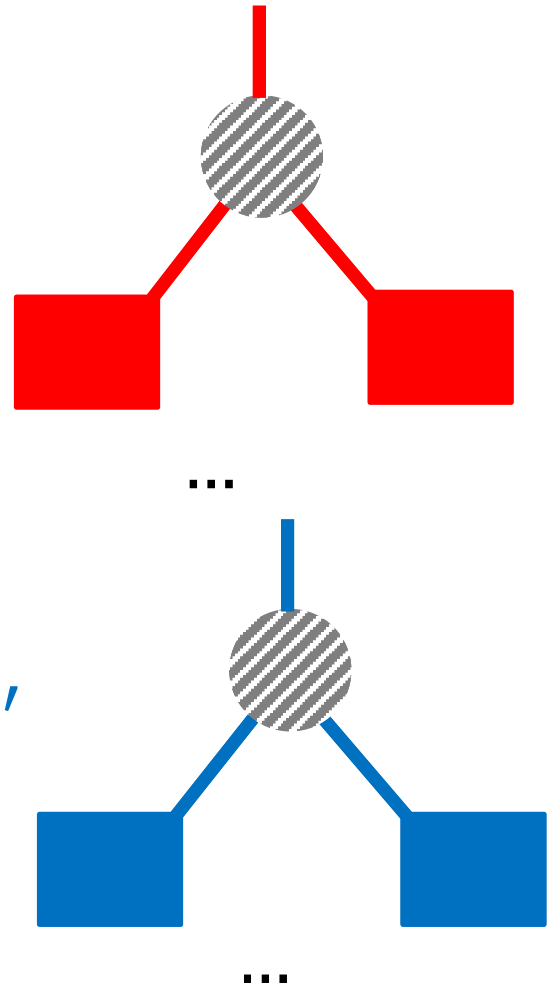
Input: x



Challenge: Mix-and-Match Attack

Input: x

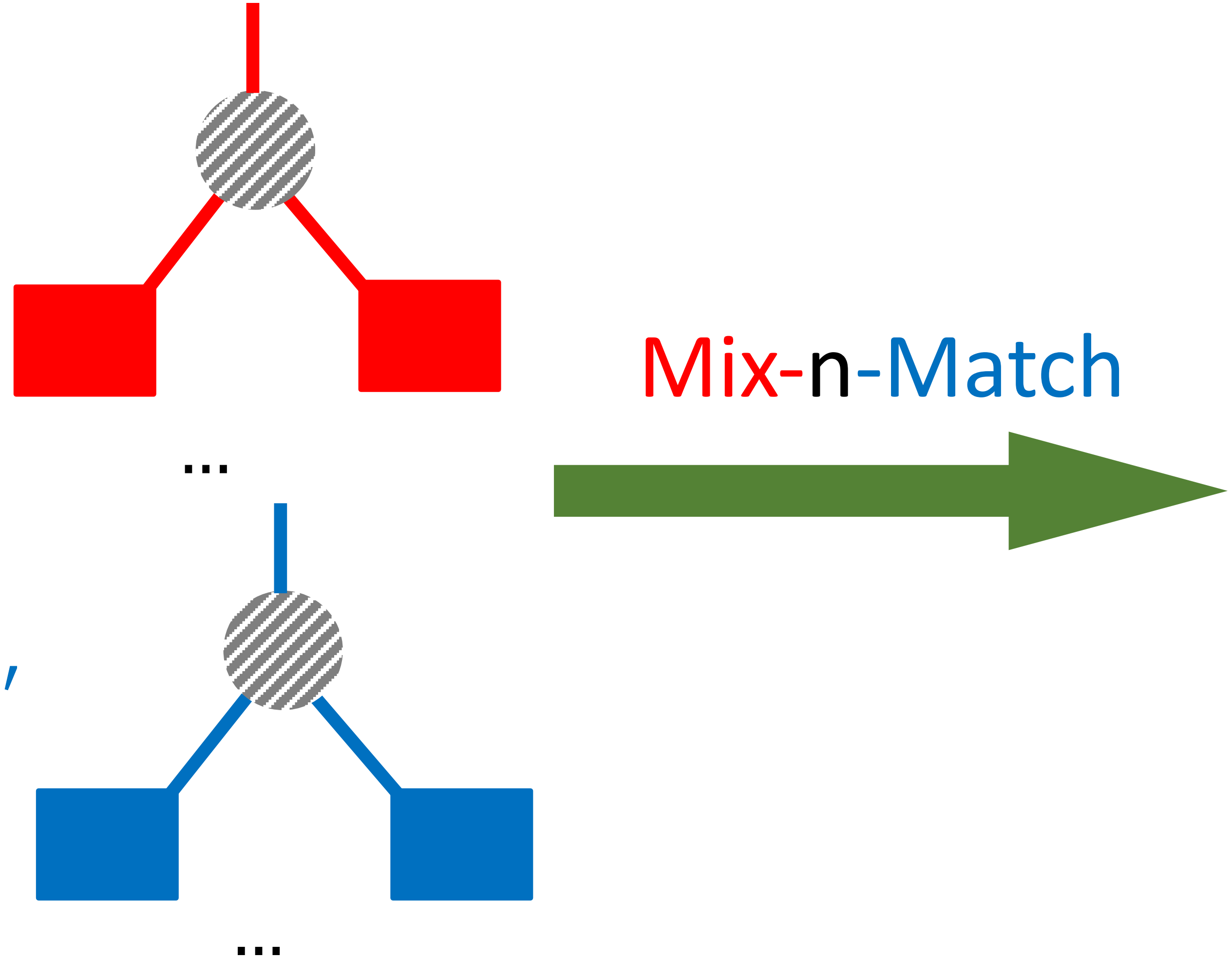
Input: x'



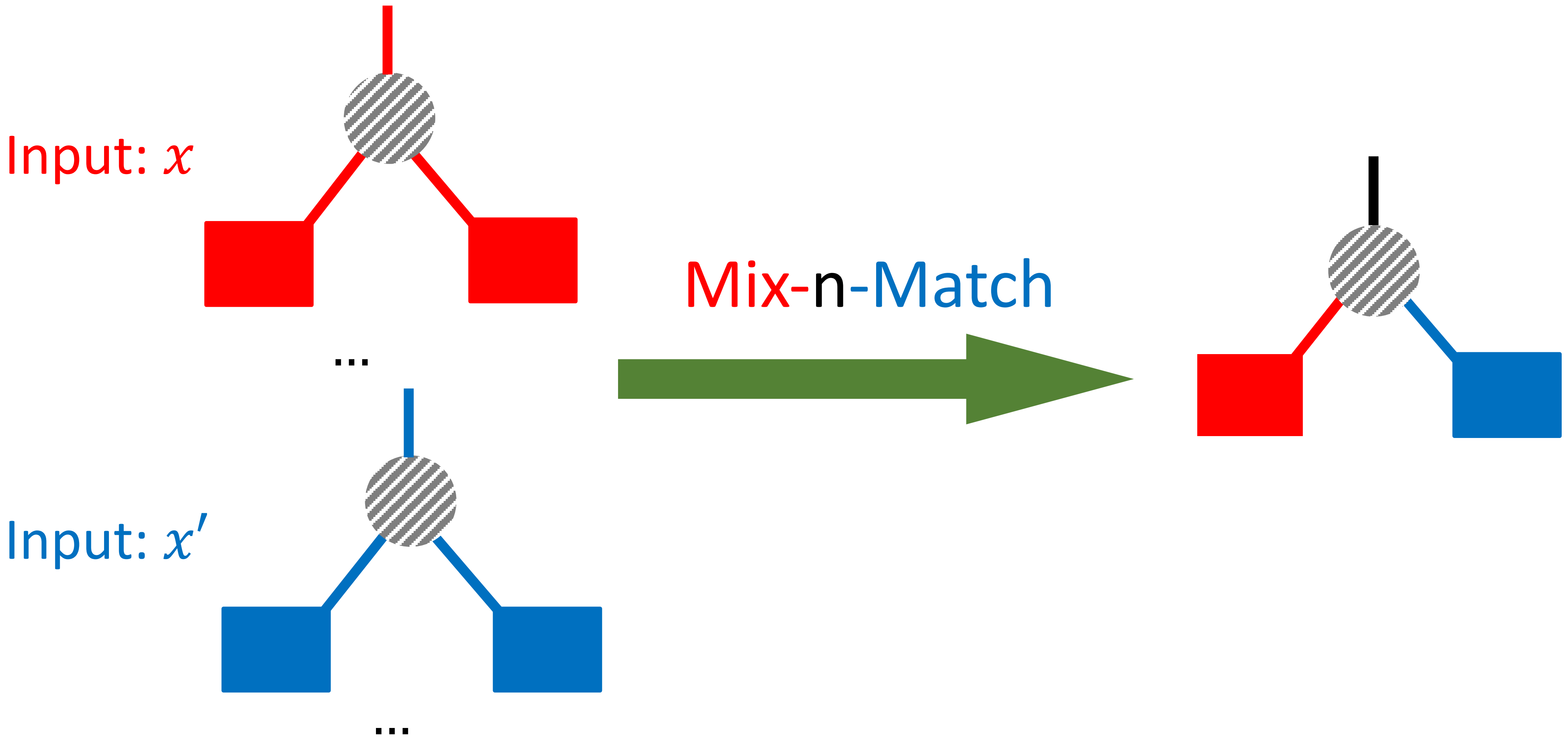
Challenge: Mix-and-Match Attack

Input: x

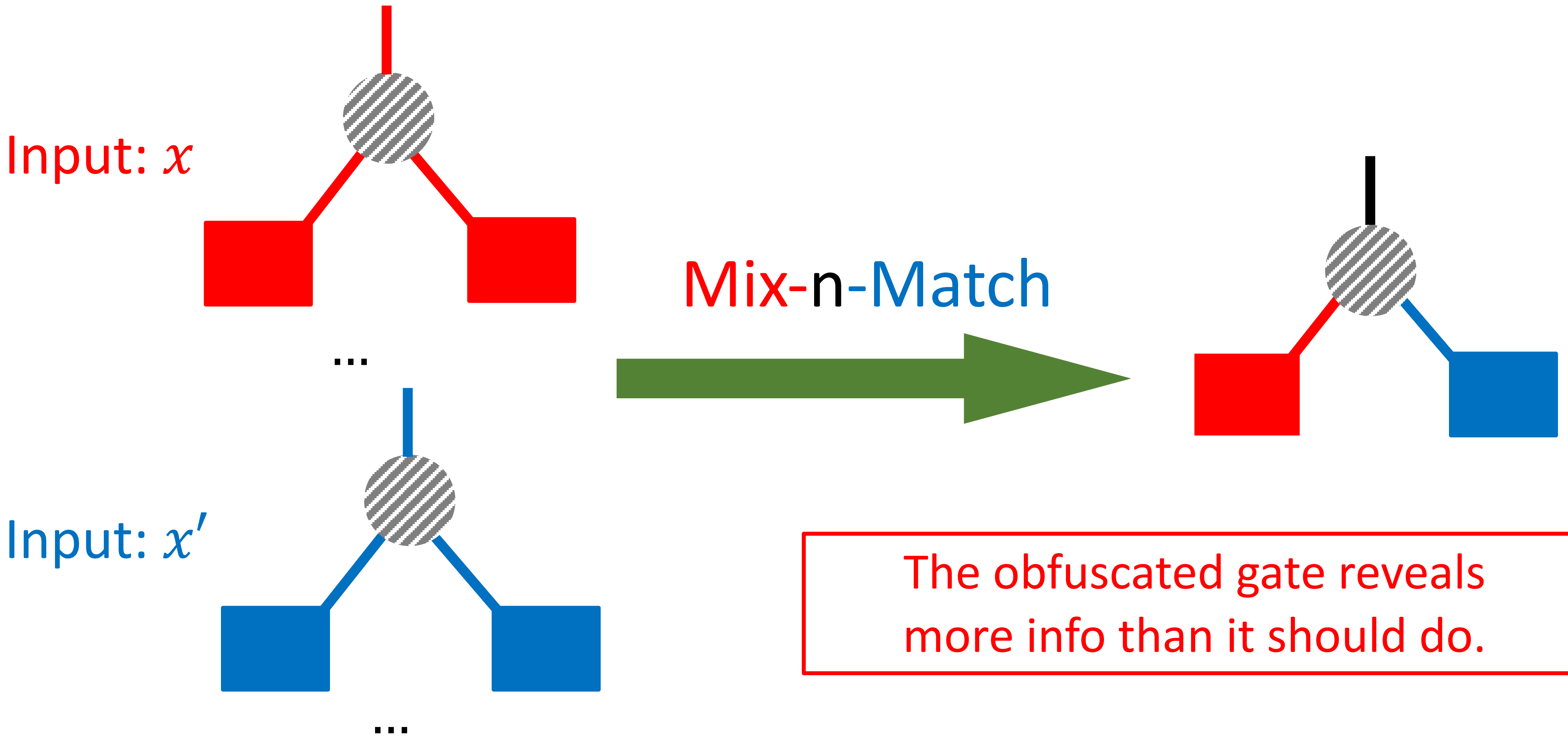
Input: x'



Challenge: Mix-and-Match Attack



Challenge: Mix-and-Match Attack



The obfuscated gate reveals more info than it should do.

$$\frac{C_g(ct_l, ct_r, input)}{\text{Check consistency w.r.t input}}$$

....

New Challenge

Too Long

$$C_g(ct_l, ct_r, input)$$

Check consistency w.r.t input

....

Idea 1: Replace Input with Dependent Wires

New Challenge

Too Long

$$C_g(ct_l, ct_r, input)$$

Check consistency w.r.t input

....

Idea 1: Replace Input with Dependent Wires

New Challenge
Too Long

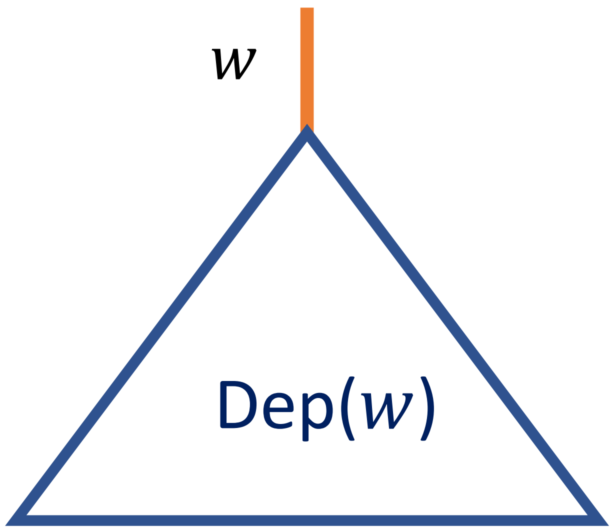
$$\frac{C_g(ct_l, ct_r, input)}{}$$

Check consistency w.r.t input

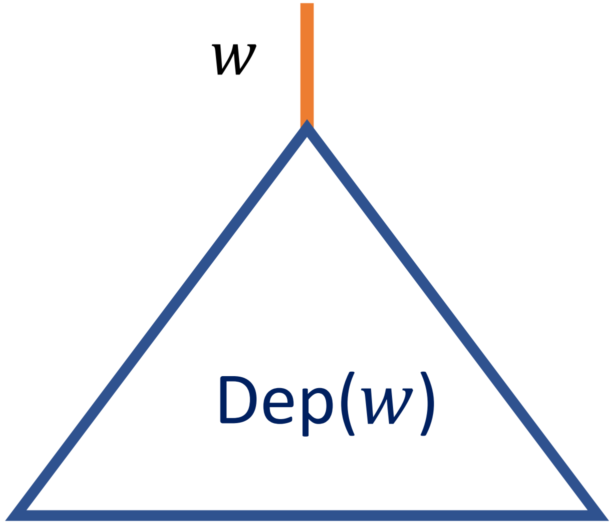
....

Gate g may not depend on the entire input
(e.g. NC^0 circuits)

Define Dependence

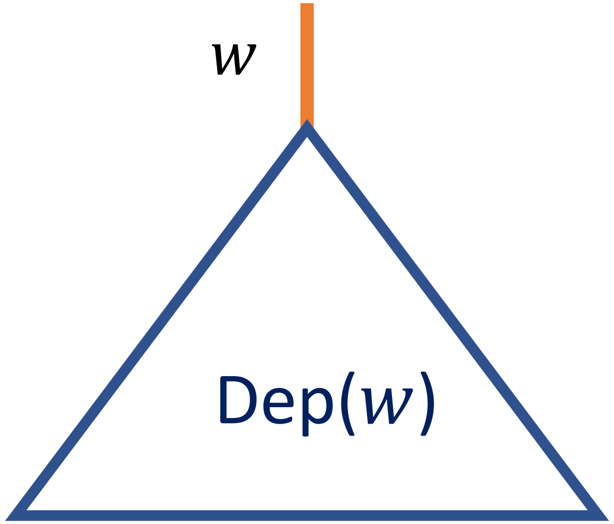


Define Dependence



Dep(w) $:=$ { all wires that w depends on }

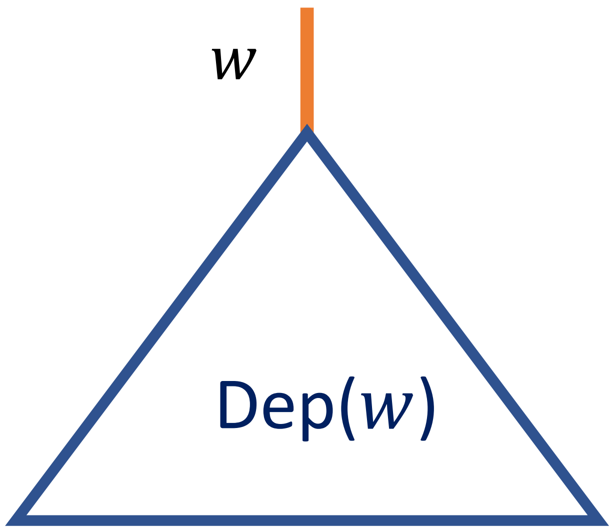
Define Dependence



$\mathrm{Dep}(w) :=$ { all wires that w depends on }

$$CT_w := \{ciphertext\ of\ k\}_{k \in \mathrm{Dep}(w)}$$

Define Dependence

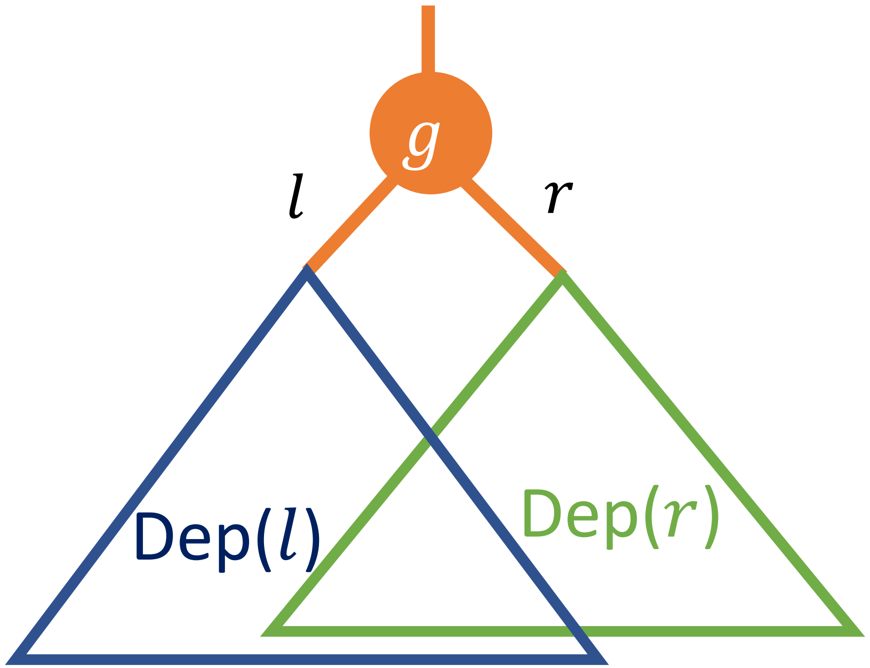


$\text{Dep}(w) :=$ { all wires that w depends on }

$$CT_w := \{ciphertext\ of\ k\}_{k \in \text{Dep}(w)}$$

(An Index Set)

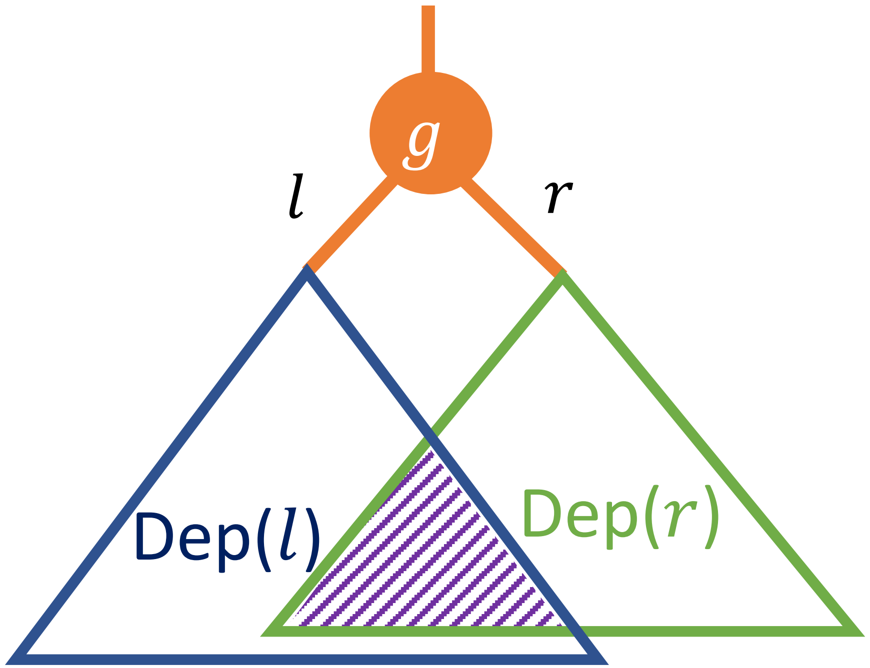
Use CT_l, CT_r in C_g



$$\frac{C_g(ct_l, ct_r, CT_l, CT_r)}{}$$

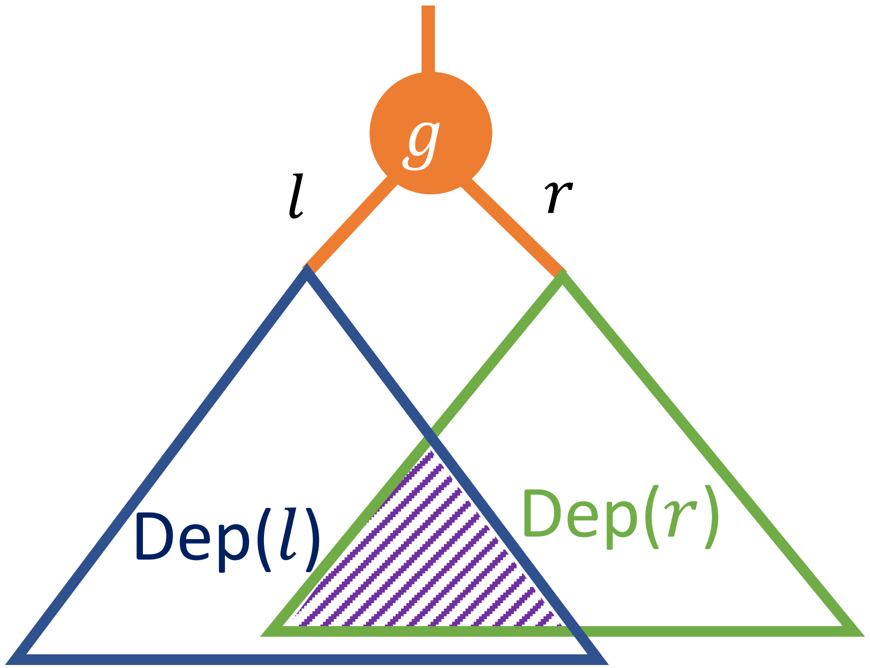
...

Use CT_l, CT_r in C_g



$$\frac{C_g(ct_l, ct_r, CT_l, CT_r)}{\ldots}$$

Use CT_l, CT_r in C_g



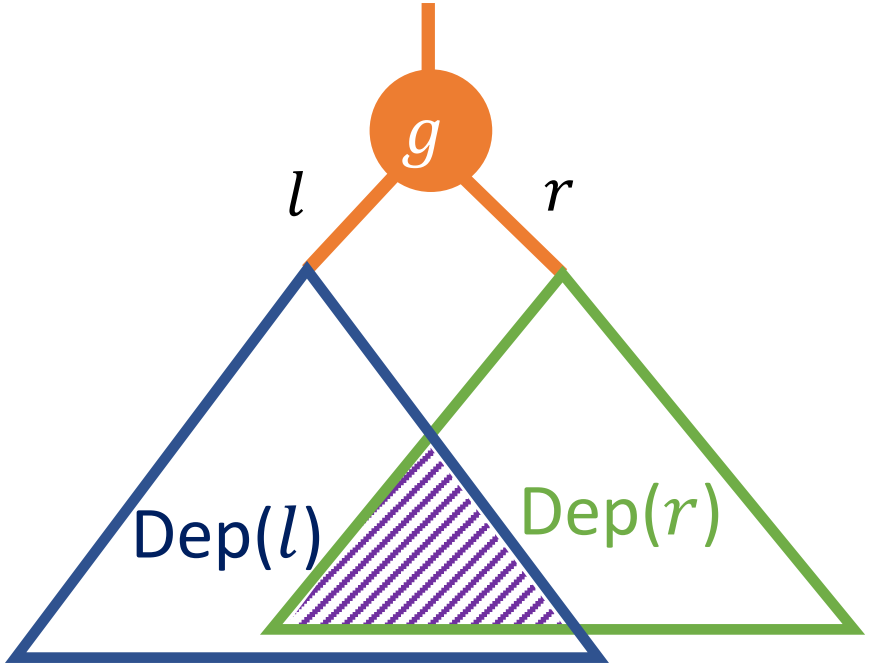
$$C_g(ct_l, ct_r, CT_l, CT_r)$$

Consistency Check:

CT_l, CT_r contains same ciphertexts in Dep(l) ∩ Dep(r)

...

Use CT_l, CT_r in C_g



New Challenge: for general circuits, still too Long

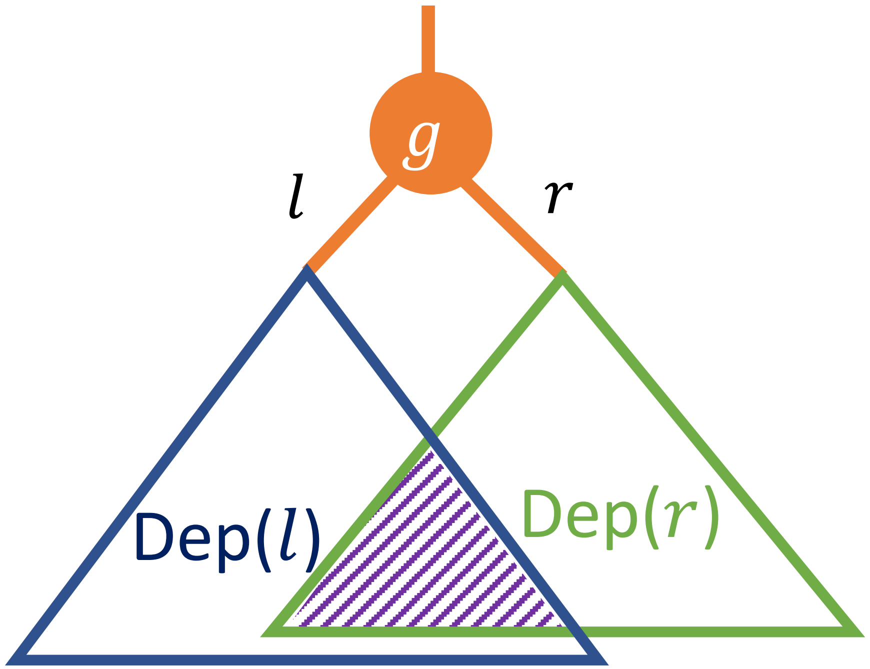
$$C_g(ct_l, ct_r, CT_l, CT_r)$$

Consistency Check:

CT_l, CT_r contains same ciphertexts in Dep(l) ∩ Dep(r)

…

Use CT_l, CT_r in C_g



$$C_g(ct_l, ct_r, CT_l, CT_r)$$

<u>**Consistency Check:**</u>

CT_l, CT_r contains same ciphertexts in Dep(l) ∩ Dep(r)

…

Idea 2: Hash CT_l, CT_r

$$\frac{C_g(ct_l, ct_r, CT_l, CT_r)}{\ldots}$$

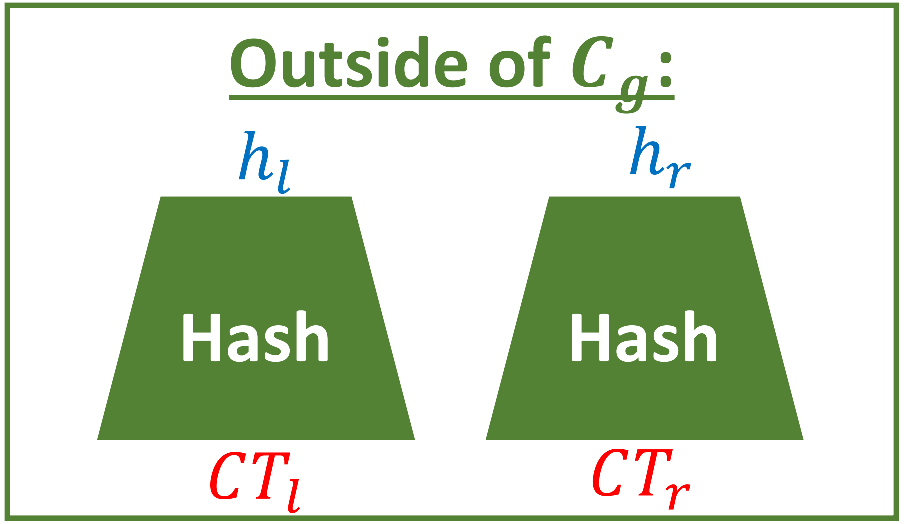
…

Idea 2: Hash CT_l, CT_r

$C_g(ct_l, ct_r, CT_l, CT_r)$

…

…

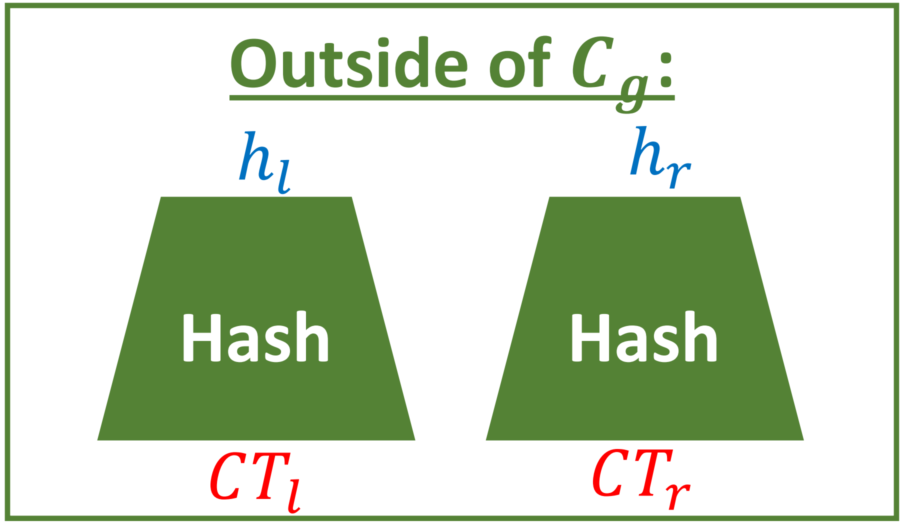


Idea 2: Hash CT_l, CT_r

$$C_g(ct_l, ct_r, \qquad\qquad)$$

…

…



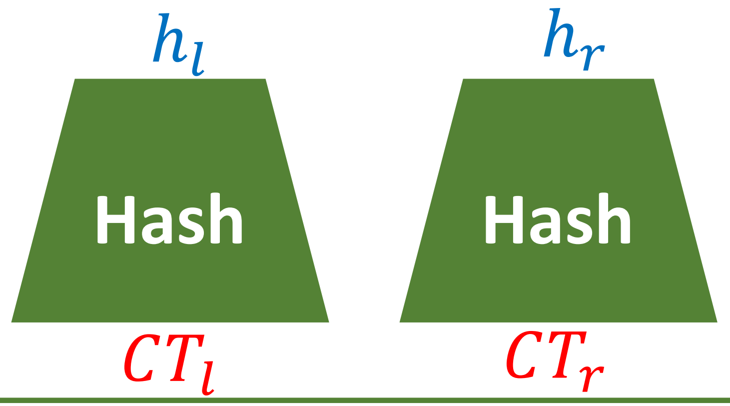
Idea 2: Hash CT_l, CT_r

$$C_g(ct_l, ct_r, \quad h_l, h_r \quad)$$

...

...

Outside of C_g:



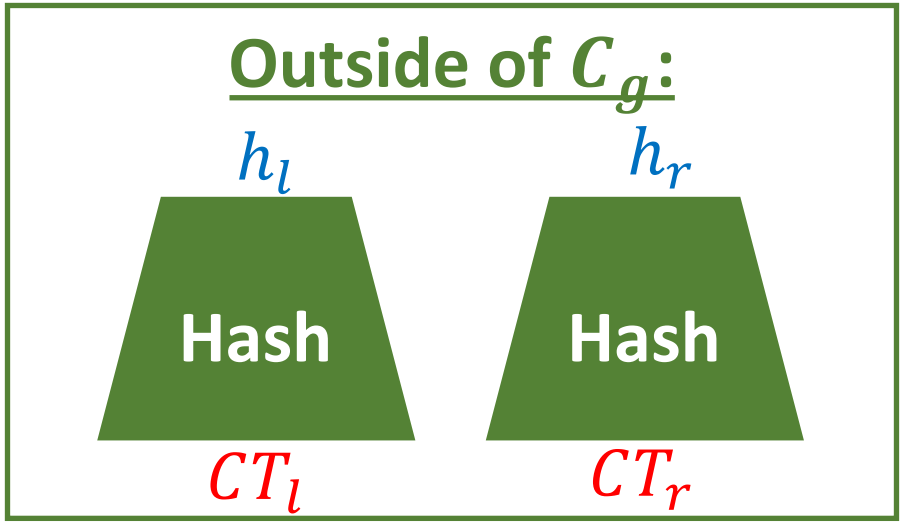
Idea 2: Hash CT_l, CT_r

$$C_g(ct_l, ct_r, \quad h_l, h_r \quad)$$

...

Check consistency of CT_l and CT_r

...

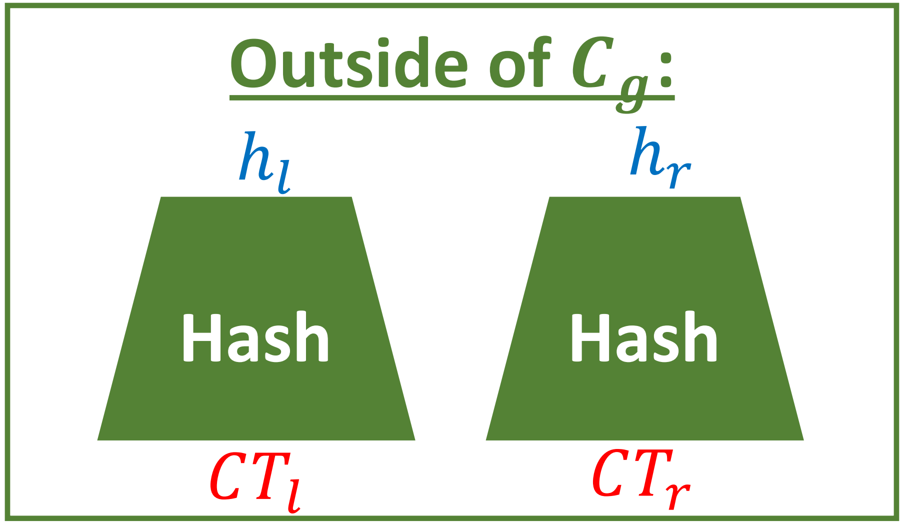


Idea 2: Hash CT_l, CT_r

$$\frac{C_g(ct_l, ct_r, \quad h_l, h_r \quad)}{\ldots}$$

Check consistency of CT_l and CT_r???

...



Idea 3: Apply SNARGs

Outside C_g:

$$h_l = Hash(CT_l)$$
$$h_r = Hash(CT_r)$$

Idea 3: Apply SNARGs

Outside C_g:

$$h_l = Hash(CT_l)$$
$$h_r = Hash(CT_r)$$

SNARGs (Succinct Cryptographic Proofs)

π : prove ∃ consistent pre-images of h_l, h_r

Secure against poly-time adversary

Idea 3: Apply SNARGs

Outside C_g: $h_l = Hash(CT_l)$
$h_r = Hash(CT_r)$

SNARGs (Succinct Cryptographic Proofs)

π : prove ∃ consistent pre-images of h_l, h_r

Secure against poly-time adversary

$$\frac{C_g(ct_l, ct_r, h_l, h_r, \pi)}{\text{Verify the proof } \pi}$$

...Decrypt, Compute, Re-encrypt...

Idea 3: Apply SNARGs

Outside C_g: $h_l = Hash(CT_l)$
$h_r = Hash(CT_r)$

SNARGs (Succinct Cryptographic Proofs)

π : prove ∃ consistent pre-images of h_l, h_r

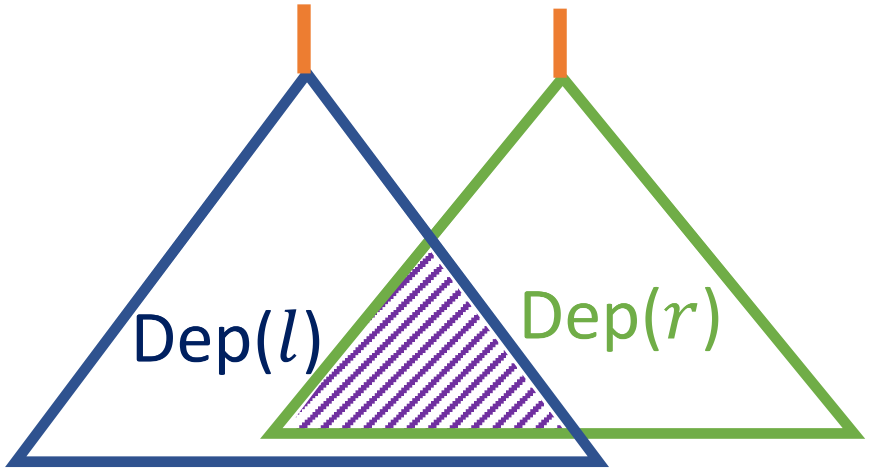
Secure against poly-time adversary

$$\frac{C_g(ct_l, ct_r, h_l, h_r, \pi)}{\text{Verify the proof } \pi}$$

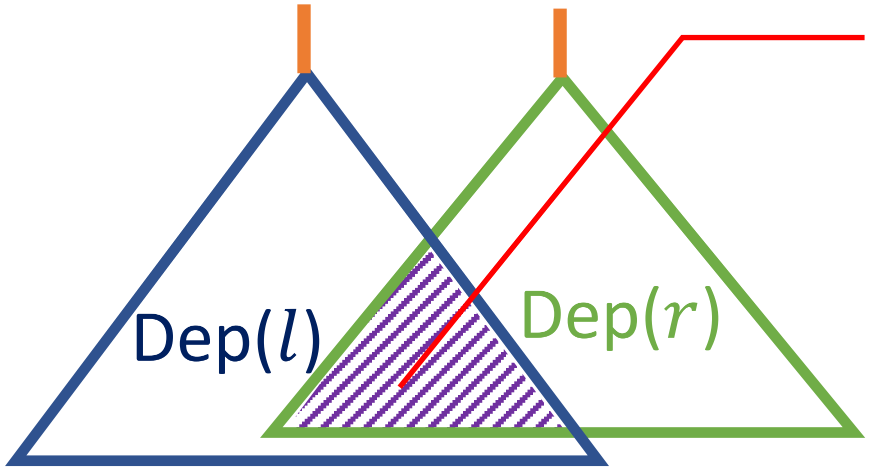
...Decrypt, Compute, Re-encrypt...

New Challenge: We need *statistical security* of SNARGs for iO.

We Use: iO-Friendly SNARGs

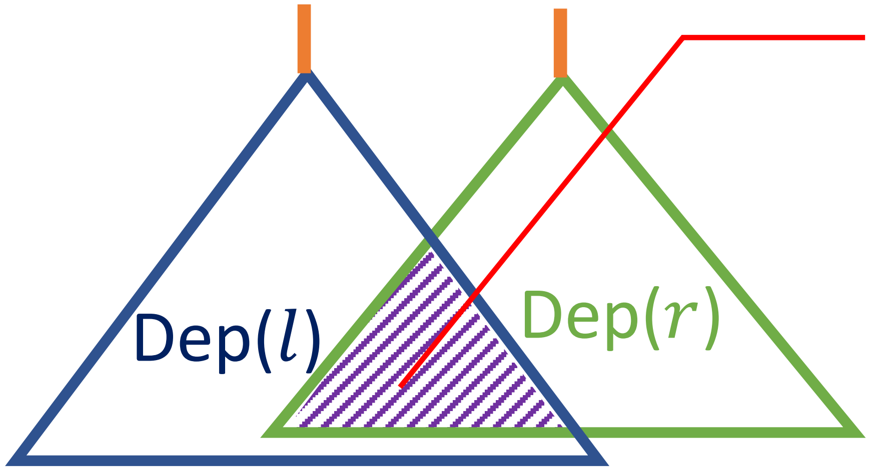


We Use: iO-Friendly SNARGs



Observation: We only care about sub-circuit

We Use: iO-Friendly SNARGs

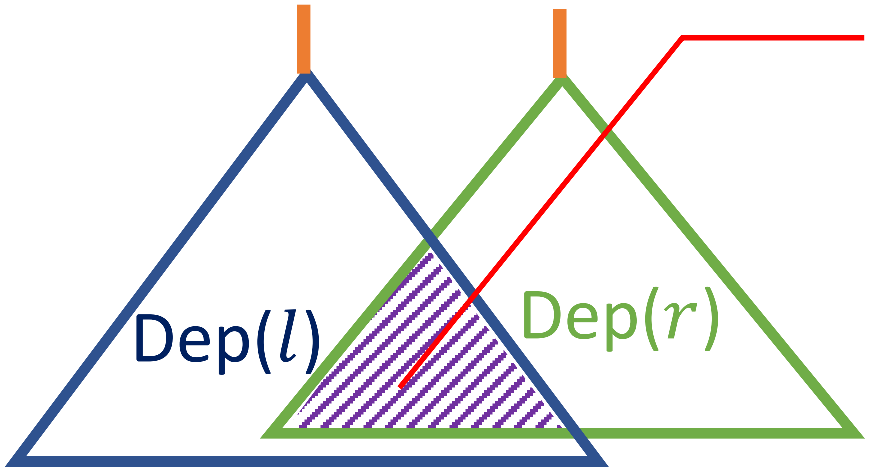


Observation: We only care about sub-circuit

Somewhere Statistical Soundness:

If CT_l and CT_r are not *consistent in subcircuit*, then unbounded-time adversary can't cheat.

We Use: iO-Friendly SNARGs

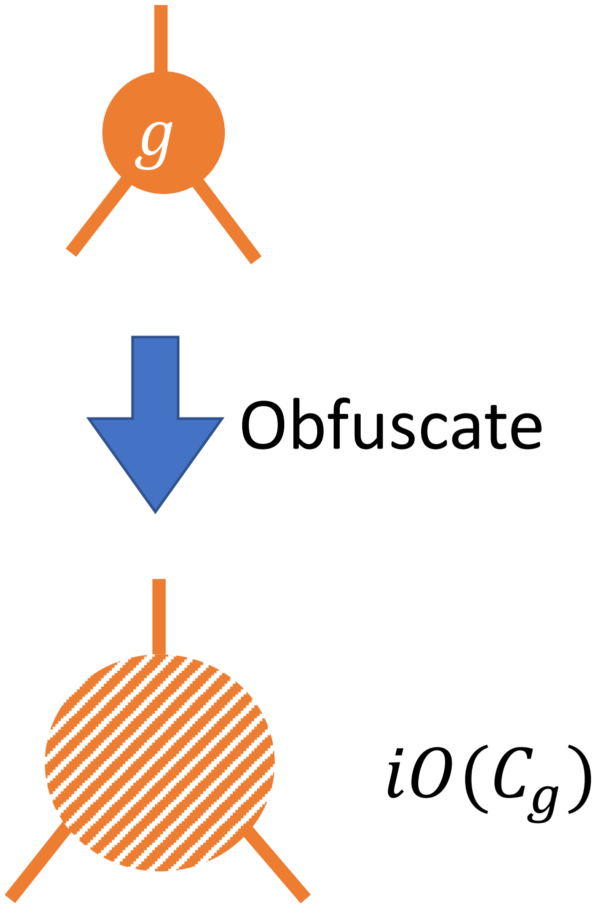


Observation: We only care about sub-circuit

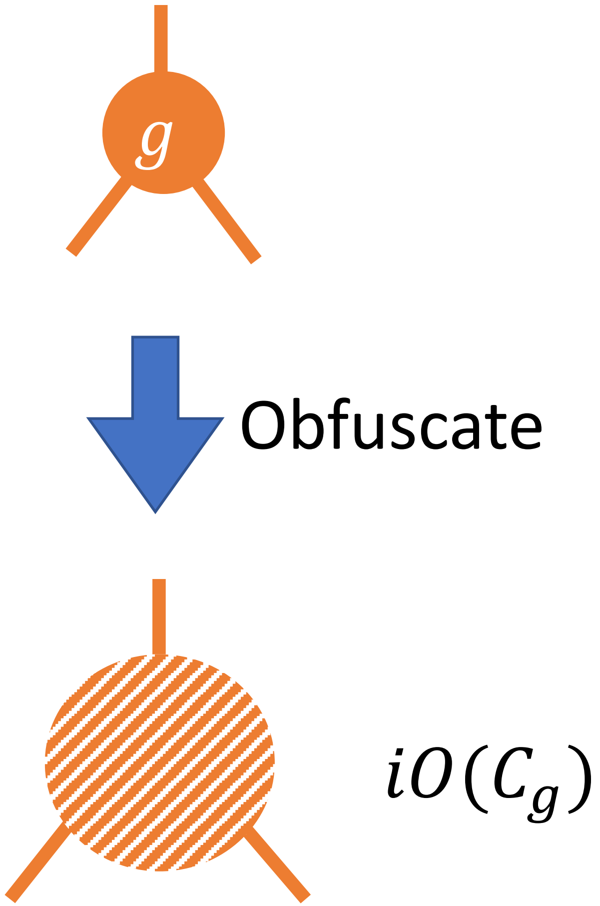
Somewhere Statistical Soundness:
If CT_l and CT_r are not *consistent in subcircuit*, then unbounded-time adversary can't cheat.

Can be constructed from [CJJ'21]

Summary



Summary

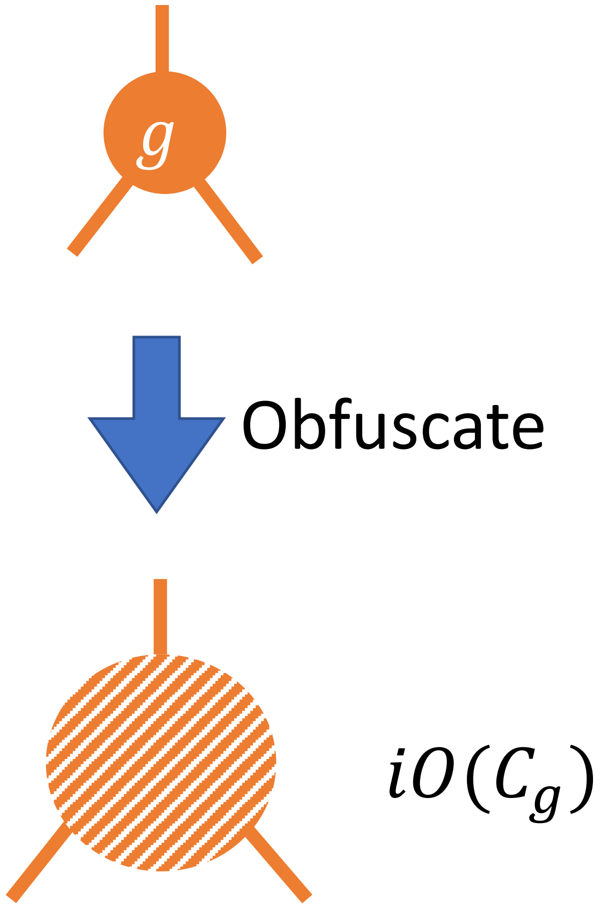


Outside C_g:

$$h_l = \text{Hash}(CT_l)$$
$$h_r = \text{Hash}(CT_r)$$

π : iO-friendly consistency proof for h_l, h_r

Summary



Outside C_g:

$$h_l = \mathrm{Hash}(CT_l)$$
$$h_r = \mathrm{Hash}(CT_r)$$

π : iO-friendly consistency proof for h_l, h_r

$$C_g(ct_l, ct_r, h_l, h_r, \pi)$$

Verify the proof π

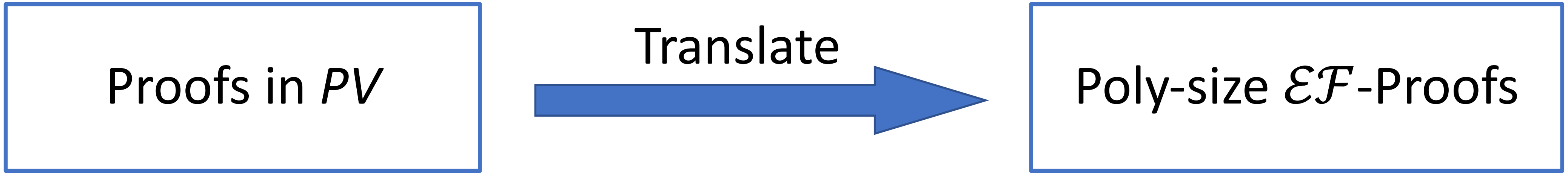
...Decrypt, Compute, Re-encrypt...

Technical Details

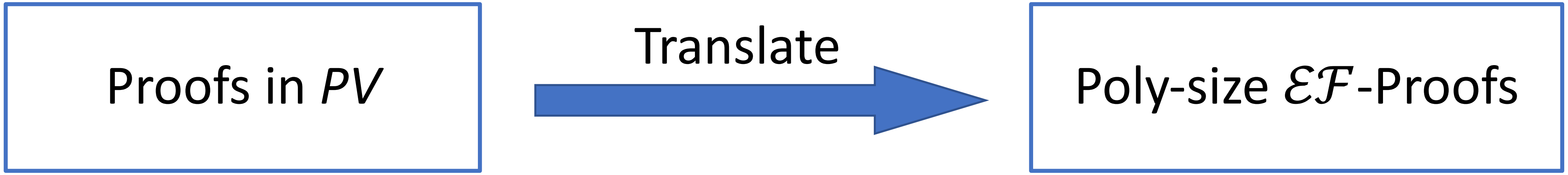
- $\mathcal{EF}$-Proofs $\Rightarrow$ local equivalence
- iO for locally equivalent ckts
- **iO for Turing machines**

Propositional Translation [Cook'75]

Propositional Translation [Cook'75]



Propositional Translation [Cook'75]



PV-Proof of
$M_1(x) = M_2(x)$

Propositional Translation [Cook'75]

Proofs in PV → Translate → Poly-size $\mathcal{EF}$-Proofs

PV-Proof of $M_1(x) = M_2(x)$ → Fix input length n →

Propositional Translation [Cook'75]

Proofs in *PV* → (Translate) → Poly-size $\mathcal{EF}$-Proofs

PV-Proof of $M_1(x) = M_2(x)$ → (Fix input length n) → $\mathcal{EF}$-Proof of $C_{1,n}(x) \leftrightarrow C_{2,n}(x)$

Propositional Translation [Cook'75]

Proofs in *PV* → Translate → Poly-size $\mathcal{EF}$-Proofs

PV-Proof of $M_1(x) = M_2(x)$ → Fix input length n → $\mathcal{EF}$-Proof of $C_{1,n}(x) \leftrightarrow C_{2,n}(x)$

($C_{b,n}(x)$: Circuit that computes M_b for input length n.)

Propositional Translation [Cook'75]

Proofs in PV → Translate → Poly-size $\mathcal{EF}$-Proofs

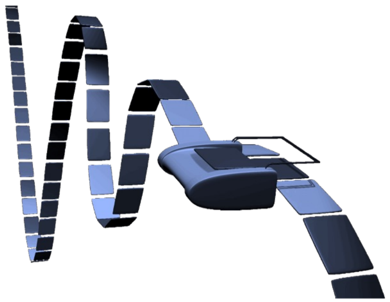
PV-Proof of $M_1(x) = M_2(x)$ → Fix input length n → $\mathcal{EF}$-Proof of $C_{1,n}(x) \leftrightarrow C_{2,n}(x)$

($C_{b,n}(x)$: Circuit that computes M_b for input length n.)

Apply iO for locally equivalent circuits?

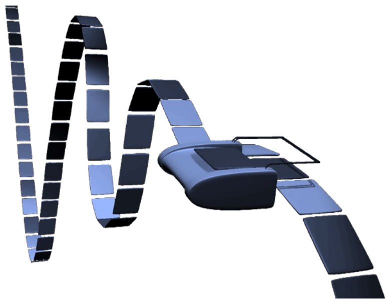
iO for Turing Machines: Initial Attempt

iO for Turing Machines: Initial Attempt



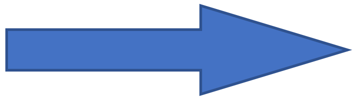
Turing Machine

iO for Turing Machines: Initial Attempt

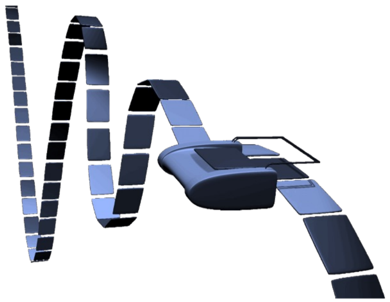


Turing Machine

$(N_0 = \lambda^{\log \lambda})$

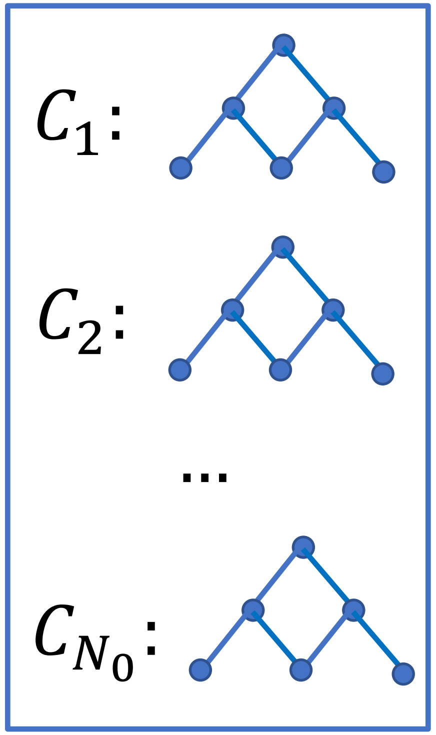


iO for Turing Machines: Initial Attempt



Turing Machine

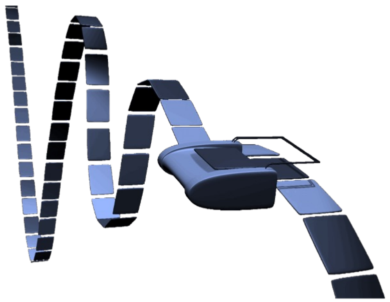
$(N_0 = \lambda^{\log \lambda})$



Ckt for every input length

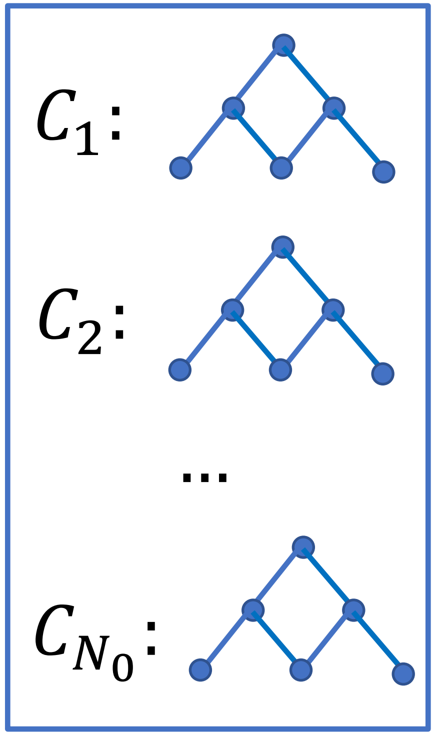
$n \leq N_0$

iO for Turing Machines: Initial Attempt



Turing Machine

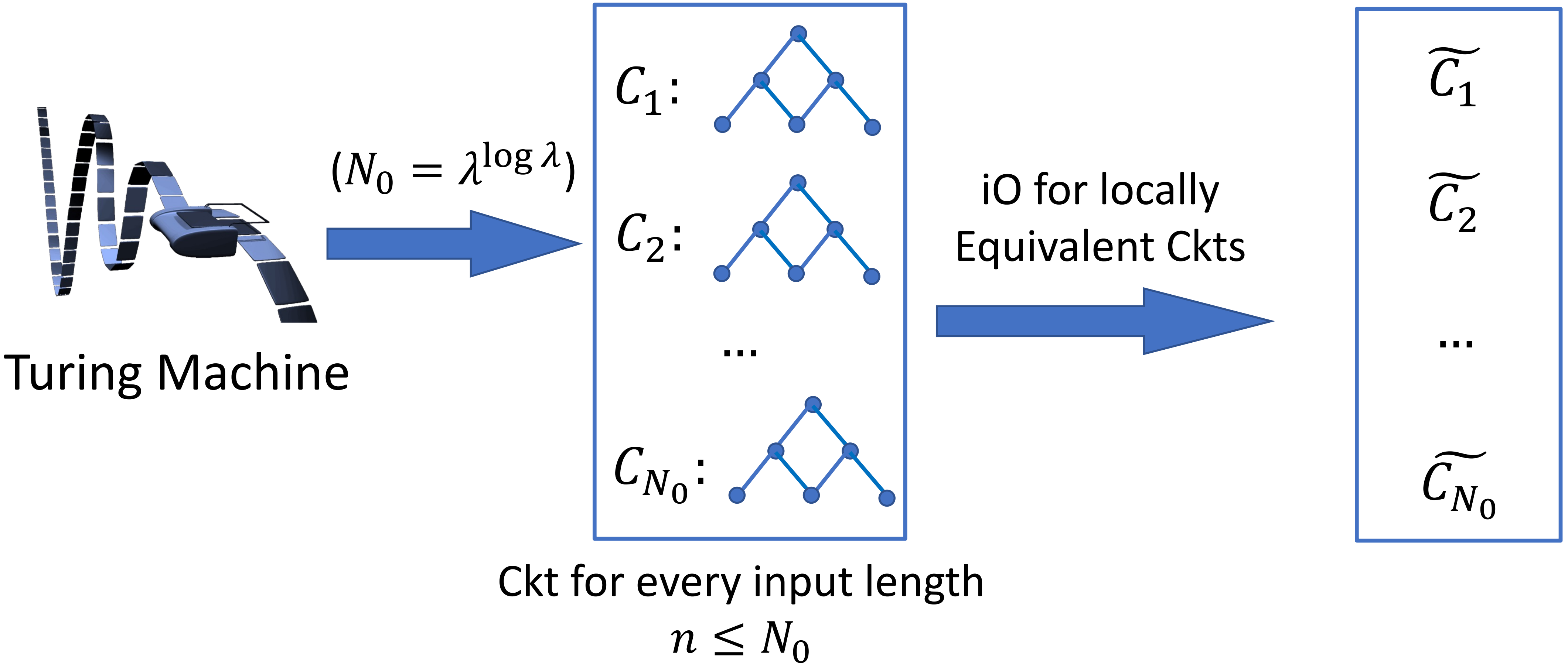
$(N_0 = \lambda^{\log \lambda})$



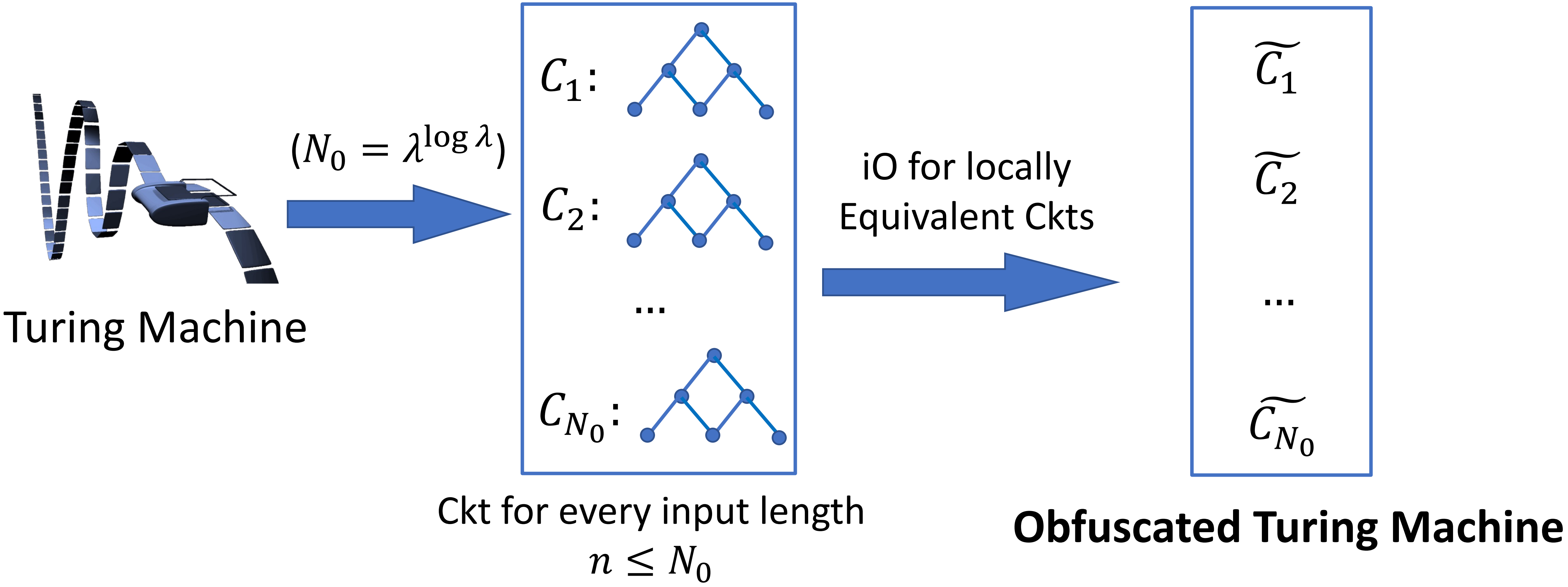
Ckt for every input length
$n \leq N_0$

iO for locally
Equivalent Ckts

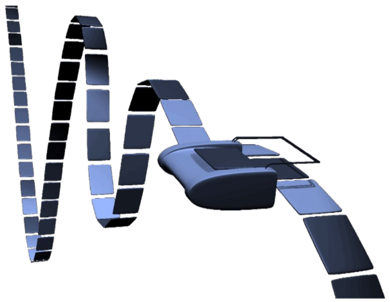
iO for Turing Machines: Initial Attempt



iO for Turing Machines: Initial Attempt



iO for Turing Machines: Initial Attempt



Turing Machine

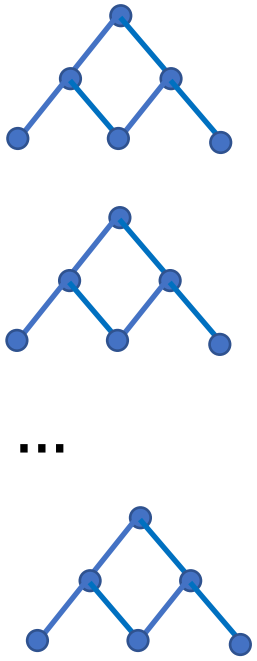
$(N_0 = \lambda^{\log \lambda})$

C_1:

C_2:

...

C_{N_0}:



Ckt for every input length $n \leq N_0$

iO for locally Equivalent Ckts

$\widetilde{C_1}$

$\widetilde{C_2}$

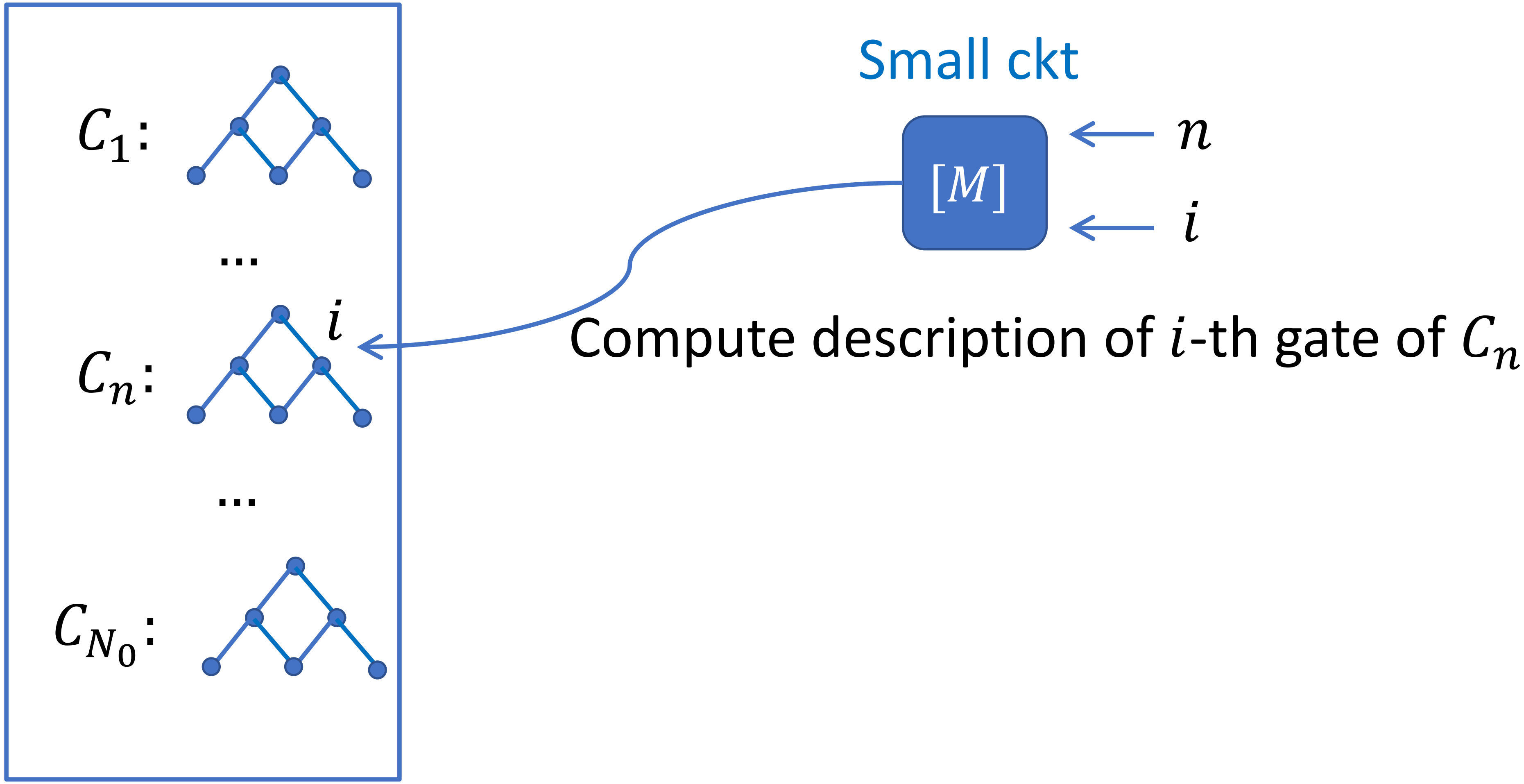
...

$\widetilde{C_{N_0}}$

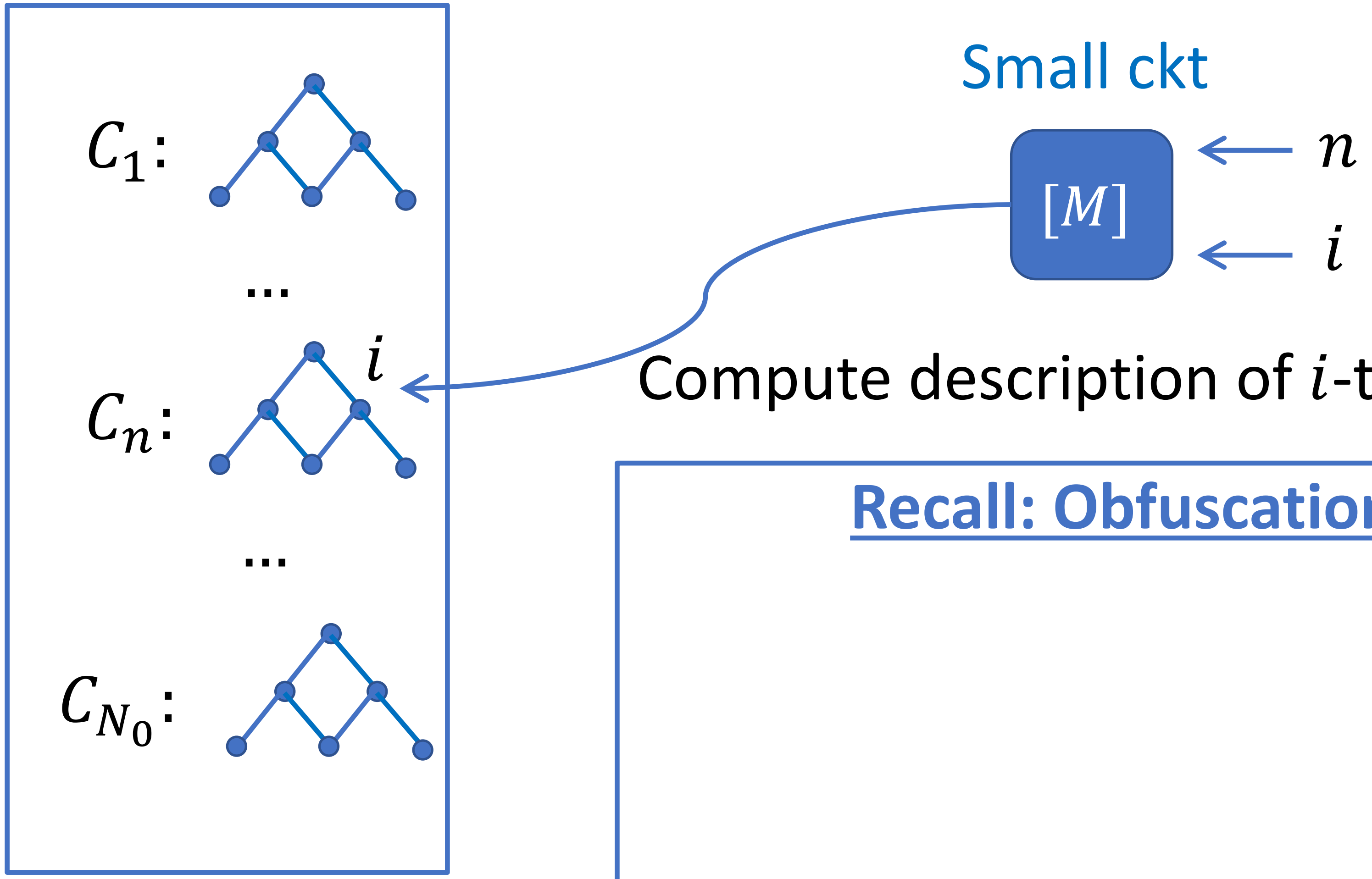
Obfuscated Turing Machine

Challenge: Obfuscation time is super-poly!

Leverage Uniform Description



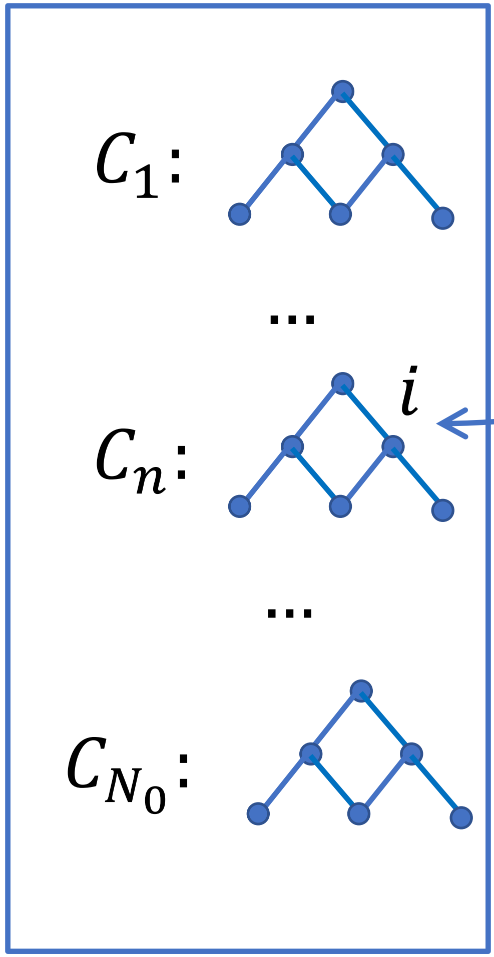
Leverage Uniform Description

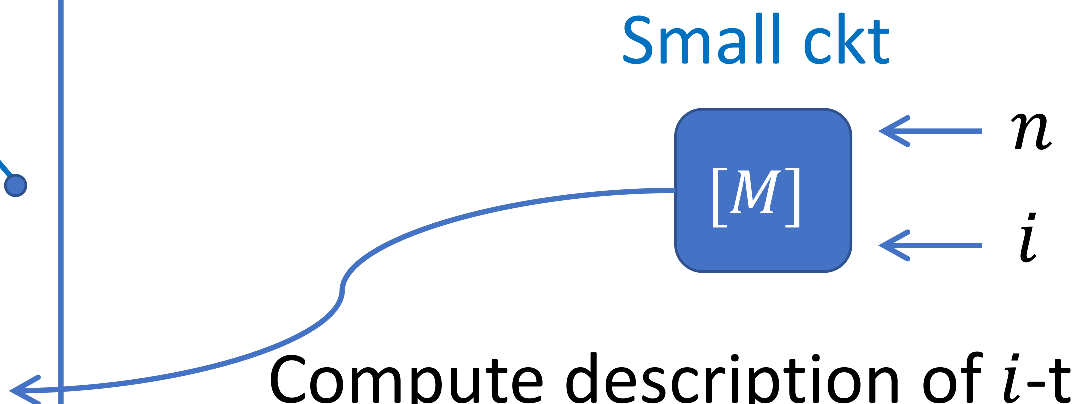


Compute description of i-th gate of C_n

Recall: Obfuscation of C_n

Leverage Uniform Description





Compute description of i-th gate of C_n

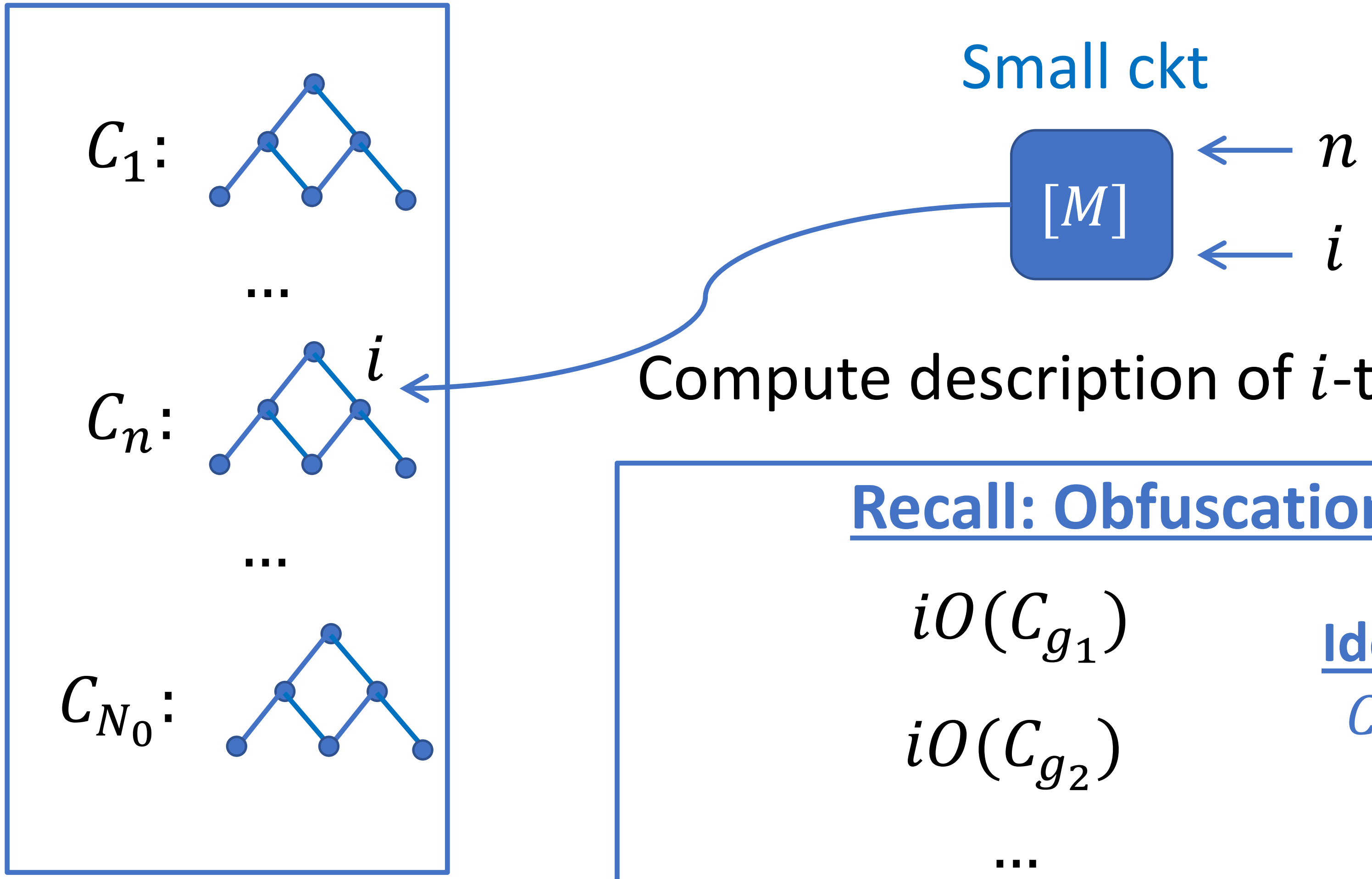
Recall: Obfuscation of C_n

$iO(C_{g_1})$

$iO(C_{g_2})$

...

Leverage Uniform Description



Compute description of i-th gate of C_n

Recall: Obfuscation of C_n

$iO(C_{g_1})$

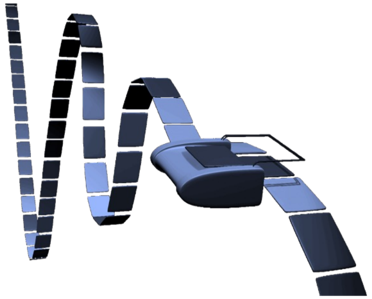
$iO(C_{g_2})$

...

Idea: Generate C_g on-the-fly using $[M]$

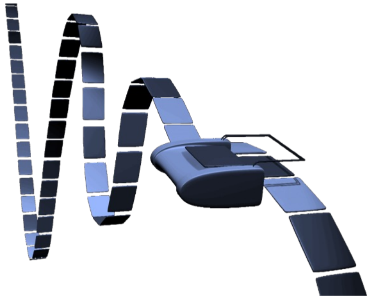
Efficient Construction

Efficient Construction

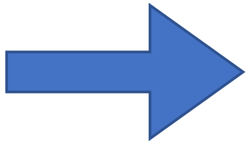


Turing Machine

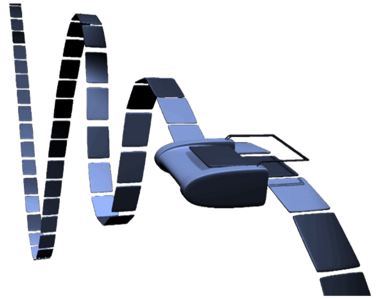
Efficient Construction



Turing Machine



Efficient Construction

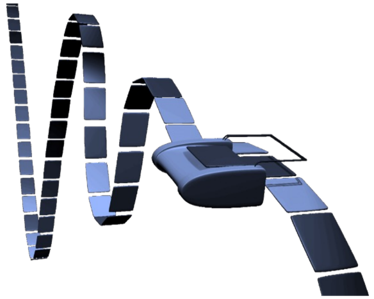


Turing Machine

UC

"Uniform Version" C_g

Efficient Construction



Turing Machine

UC

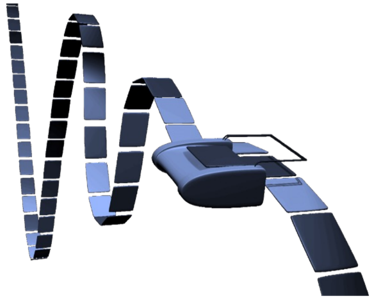
"Uniform Version" C_g

<u>$UC(n, i, input')$</u>

- Get description of i-th gate of C_n:

- Compute $C_g(input')$

Efficient Construction



Turing Machine

UC

"Uniform Version" C_g

iO for Ckt

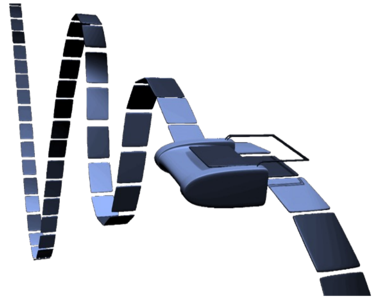
$\underline{UC(n, i, input')}$

- Get description of i-th gate of C_n:

  $g \leftarrow [M] \leftarrow n, i$

- Compute $C_g(input')$

Efficient Construction



Turing Machine

iO for Ckt

UC → $iO(UC)$

"Uniform Version" C_g

Obfuscated Turing Machine

<u>$UC(n, i, input')$</u>

- Get description of i-th gate of C_n:

  $g \leftarrow [M] \leftarrow n, i$

- Compute $C_g(input')$

Summary & Future Directions

Summary & Future Directions

Inference Rules in
Logic systems for
Proving Equivalence

Summary & Future Directions

Inference Rules in
Logic systems for
Proving Equivalence

↔

Summary & Future Directions

Inference Rules in
Logic systems for
Proving Equivalence

Techniques to argue
Indistinguishability for iO

Summary & Future Directions

Inference Rules in
Logic systems for
Proving Equivalence

↔

Techniques to argue
Indistinguishability for iO

$\mathcal{EF}$ / PV

Summary & Future Directions

Inference Rules in
Logic systems for
Proving Equivalence

⟺

Techniques to argue
Indistinguishability for iO

$\mathcal{EF}$ / PV

⟺

Summary & Future Directions

Inference Rules in
Logic systems for
Proving Equivalence

↔

Techniques to argue
Indistinguishability for iO

$\mathcal{EF}$ / PV

↔

Local Equivalence

Summary & Future Directions

Inference Rules in **Logic systems** for Proving Equivalence	⟷	Techniques to argue Indistinguishability for iO
$\mathcal{EF}$ / PV	⟷	Local Equivalence
ZFC (Zermelo-Fraenkel set theory with axiom of Choice)	⟷	?

Thank you!

Q & A