Indistinguishability Obfuscation via Mathematical Proofs of Equivalence

Abhishek Jain
Johns Hopkins University

Zhengzhong Jin
MIT

Indistinguishability Obfuscation (iO)

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

C

(Circuit/Turing Machine)

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();
```

C'

Indistinguishability Obfuscation (iO)

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

C

(Circuit/Turing Machine)

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();
```

C'

Preserve Functionality: $\forall\, x, C'(x) = C(x)$

Indistinguishability Security

Indistinguishability Security

For any C_0, C_1 if $\forall\, x\ C_0(x) = C_1(x)$

$$iO(1^\lambda, C_0) \approx_c iO(1^\lambda, C_1)$$

(λ : Security Parameter)

Indistinguishability Security

For any C_0, C_1 if $\forall\, x\; C_0(x) = C_1(x)$

$$iO(1^\lambda, C_0) \approx_c iO(1^\lambda, C_1)$$

(λ : Security Parameter)

Non-falsifiability

$$\xrightarrow{C_0,\, C_1}$$

$$\xleftarrow{iO(C_b)}$$

$$\xrightarrow{b'}$$

Indistinguishability Security

For any ${\color{red}C_0}, {\color{blue}C_1}$ if $\forall\, x\ {\color{red}C_0(x)} = {\color{blue}C_1(x)}$

$$iO(1^\lambda, {\color{red}C_0}) \approx_c iO(1^\lambda, {\color{blue}C_1}) \qquad (\lambda : \text{Security Parameter})$$

Non-falsifiability

$\xrightarrow{{\color{red}C_0}, {\color{blue}C_1}}$

$\xleftarrow{iO(C_b)}$

$\xrightarrow{{\color{red}b'}}$

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

Indistinguishability Security

For any C_0, C_1 if $\forall\, x\ C_0(x) = C_1(x)$

$$iO(1^\lambda, C_0) \approx_c iO(1^\lambda, C_1)$$

(λ : Security Parameter)

<u>Non-falsifiability</u>

$\xrightarrow{C_0, C_1}$

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

$\xleftarrow{iO(C_b)}$

$\xrightarrow{b'}$

Check $\forall\, x\ C_0(x) = C_1(x) \wedge b = b'$
inefficiently

Can we build iO?

Can we build iO?

- A long line of works:

  [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?

- A long line of works:

  [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]…

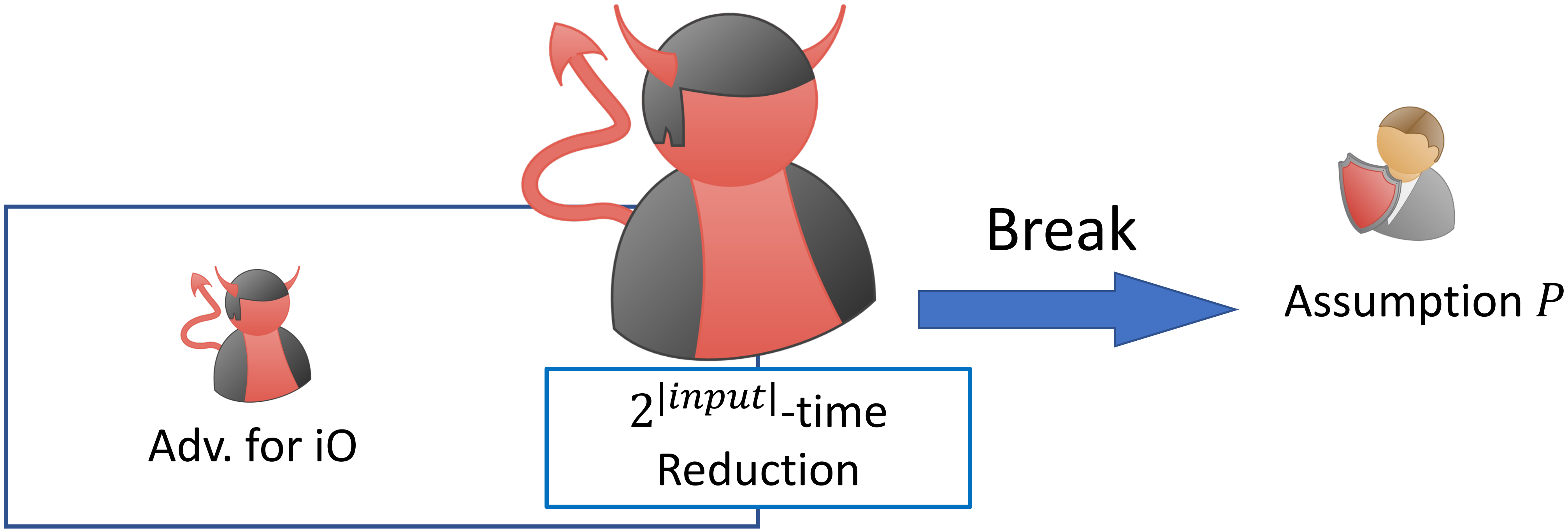
- **iO from Well-Founded Assumptions** [Jain-Lin-Sahai'20]

  Based on **Sub-exponential Security** of Learning with Errors, and Learning Parity with Noise and more…

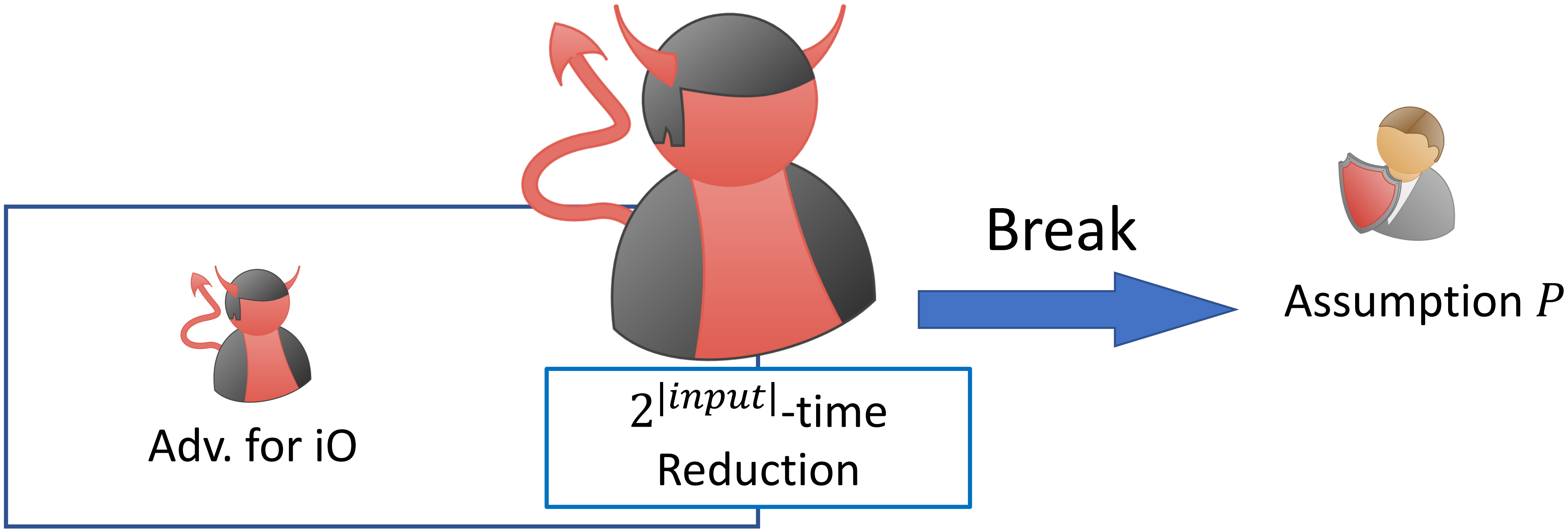
$2^{|input|}$-Loss in Reduction

Adv. for iO

$2^{|input|}$-Loss in Reduction

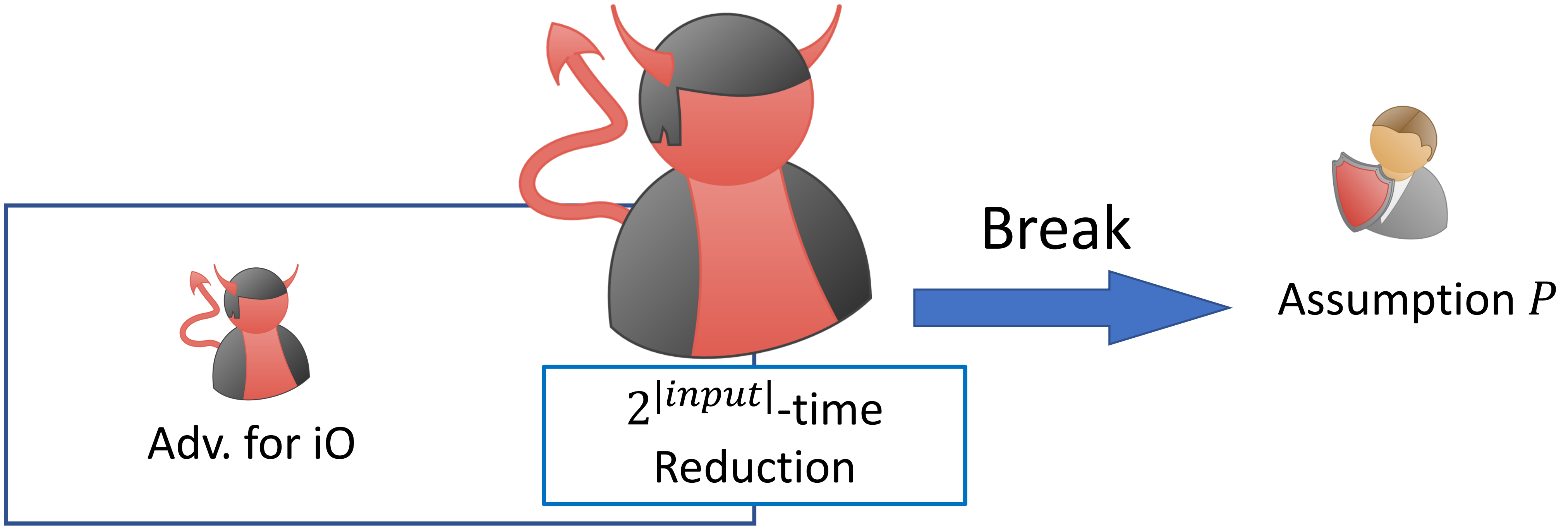


$2^{|input|}$-Loss in Reduction



Assume 2^{λ^c}-Security of P & set $2^{\lambda^c} > 2^{|input|}$

$2^{|input|}$-Loss in Reduction



Assume 2^{λ^c}-Security of P & set $2^{\lambda^c} > 2^{|input|}$

$|input| < \lambda^c$

$2^{|input|}$-Security Loss is Bad

$2^{|input|}$-Security Loss is Bad

iO for Turing Machines

M
(Turing Machine) $\longrightarrow$ M'
(Turing Machine)

$2^{|input|}$-Security Loss is Bad

M
(Turing Machine)

iO for Turing Machines $\longrightarrow$

M'
(Turing Machine)

Ideally: M' works for **unbounded input-length**

$2^{|input|}$-Security Loss is Bad

iO for Turing Machines

M (Turing Machine) $\longrightarrow$ M' (Turing Machine)

Ideally: M' works for **unbounded input-length**

<u>Prior work:</u>

Input length of M' is **bounded (since** $|input| < \lambda^c$**)**

[Bitansky-Garg-Lin-Pass-Telang’15][Canetti-Holmgren-Jain-Vaikuntanathan’15][Koppula-Lewko-Waters’15]…

iO: the “Central Hub” [Sahai-Waters’13]

Nash Equilibrium
[Bitansky-Paneth-Rosen’14]

NIZKs/SNARGs

iO

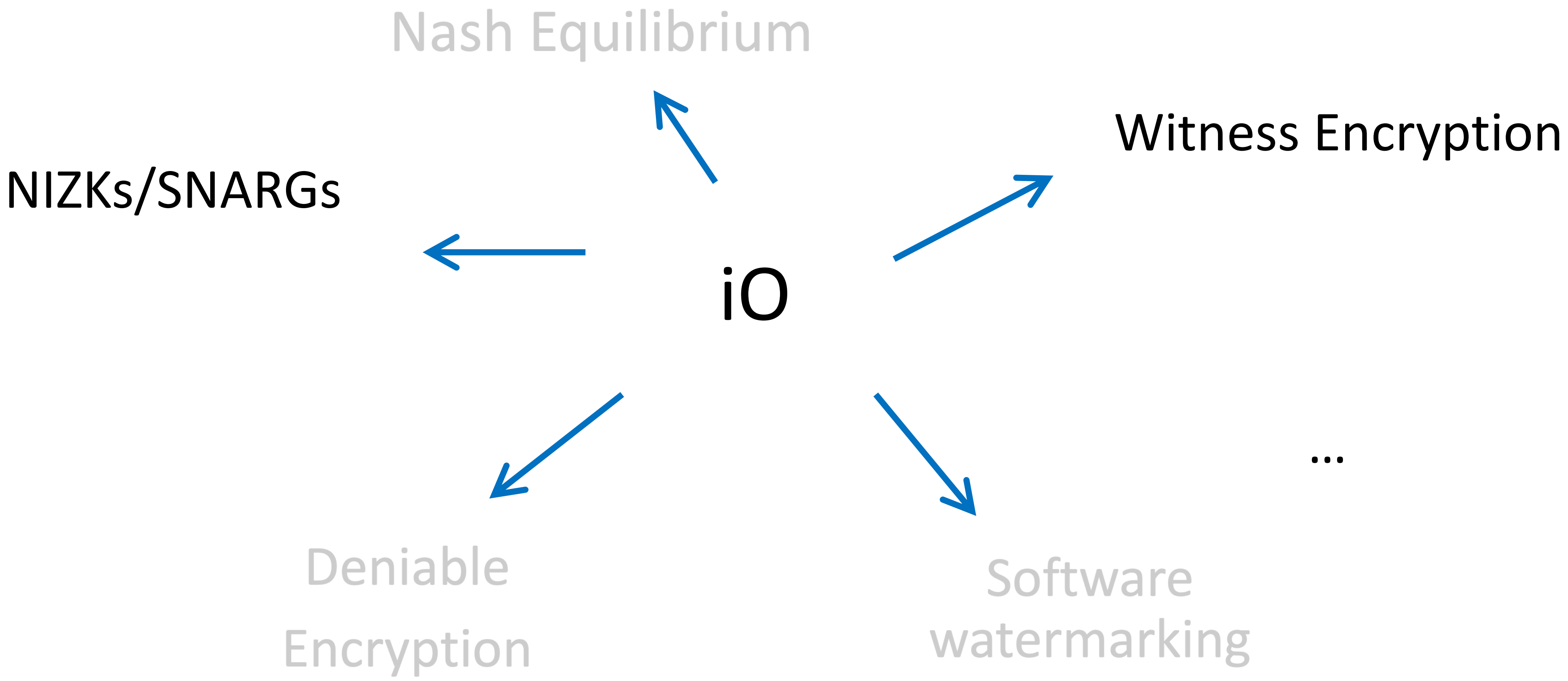
Witness Encryption

Deniable Encryption

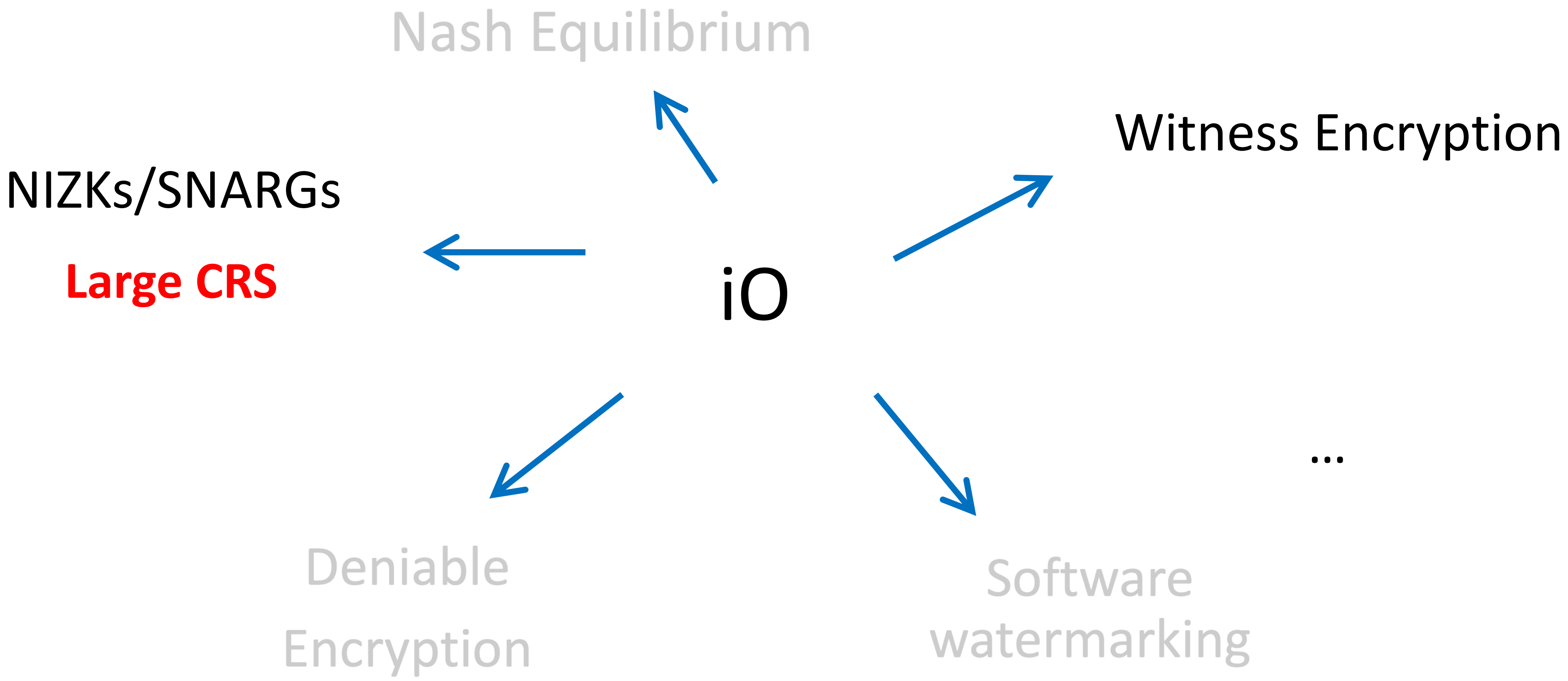
Software watermarking

…

$2^{|input|}$-Security Loss “Spreads”



$2^{|input|}$-Security Loss "Spreads"



$2^{|input|}$-Security Loss "Spreads"

Nash Equilibrium

NIZKs/SNARGs

Large CRS

iO

Witness Encryption

Large ciphertext

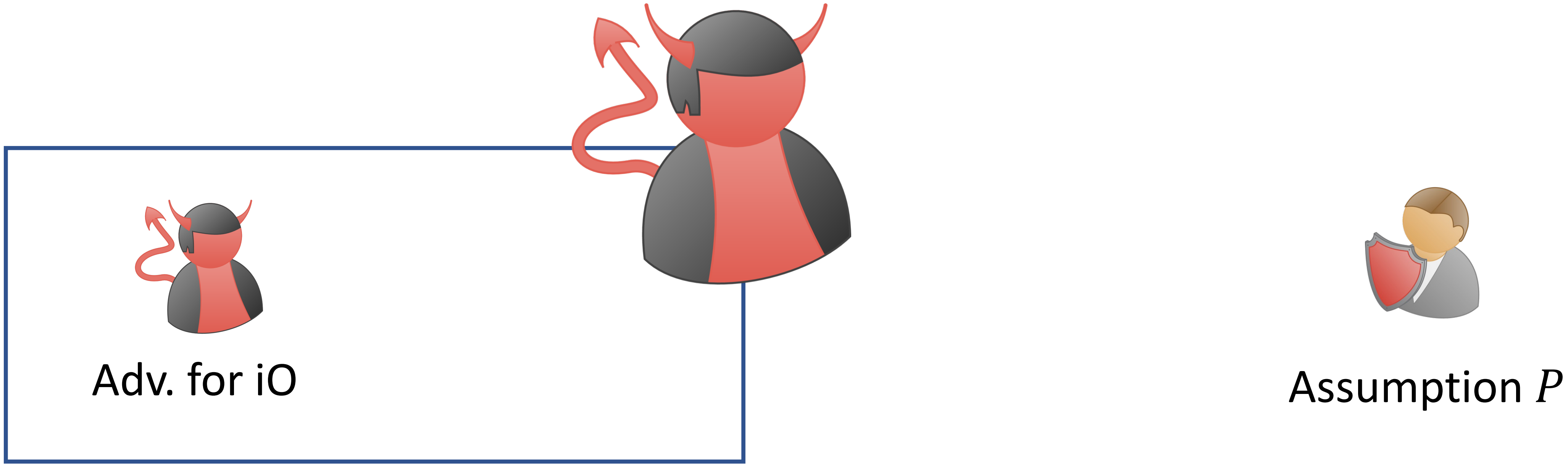
Deniable Encryption

Software watermarking

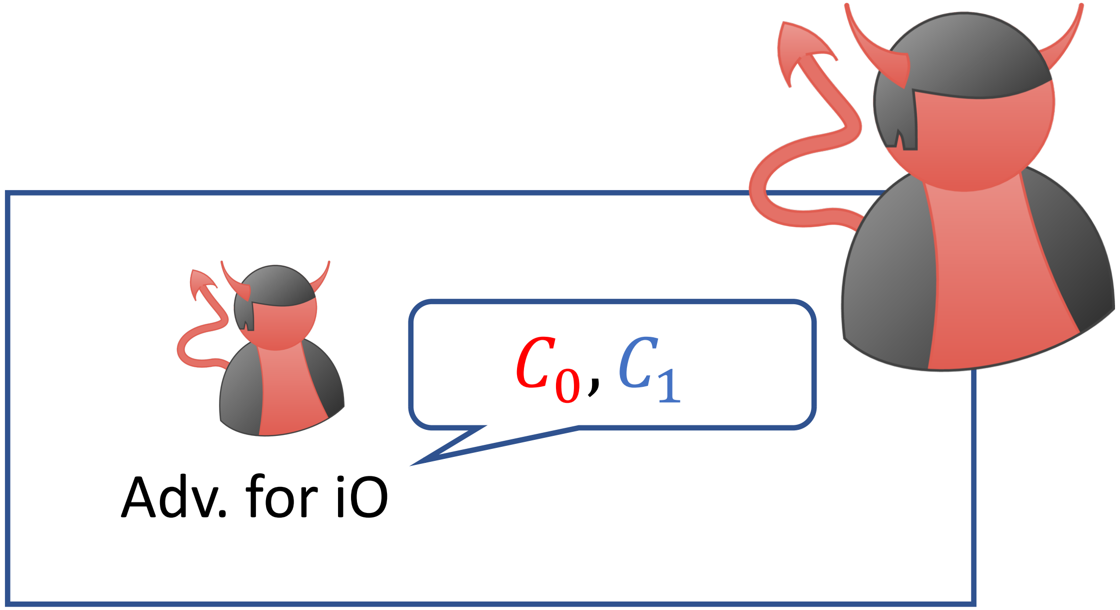
...

Question: Can we build iO with a security loss ***independent*** of the input length?

Is $2^{|input|}$-Loss Inherent? (folklore)



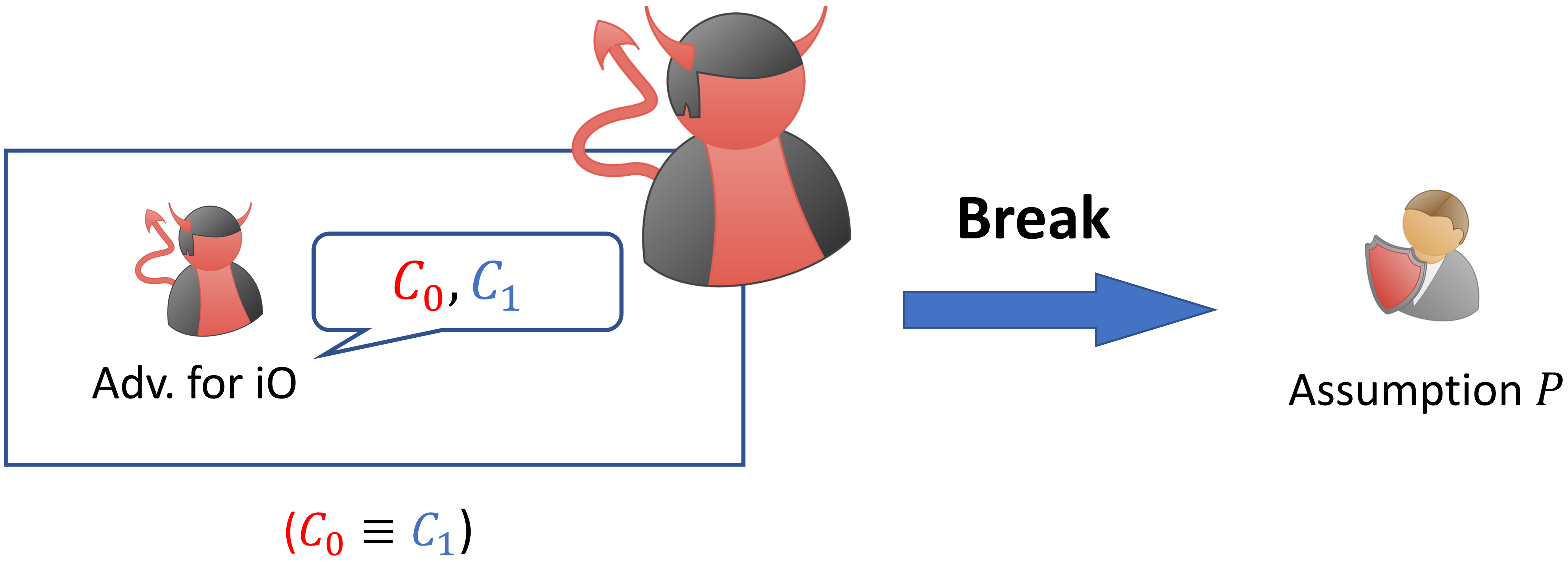
Is $2^{|input|}$-Loss Inherent? (folklore)



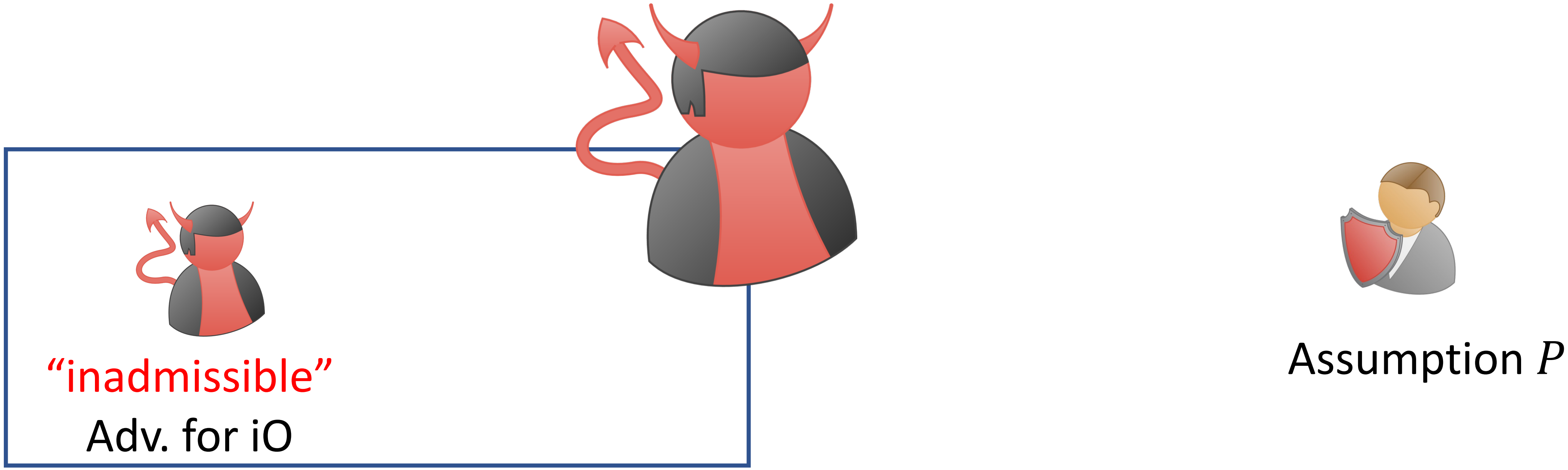
$(C_0 \equiv C_1)$

Assumption P

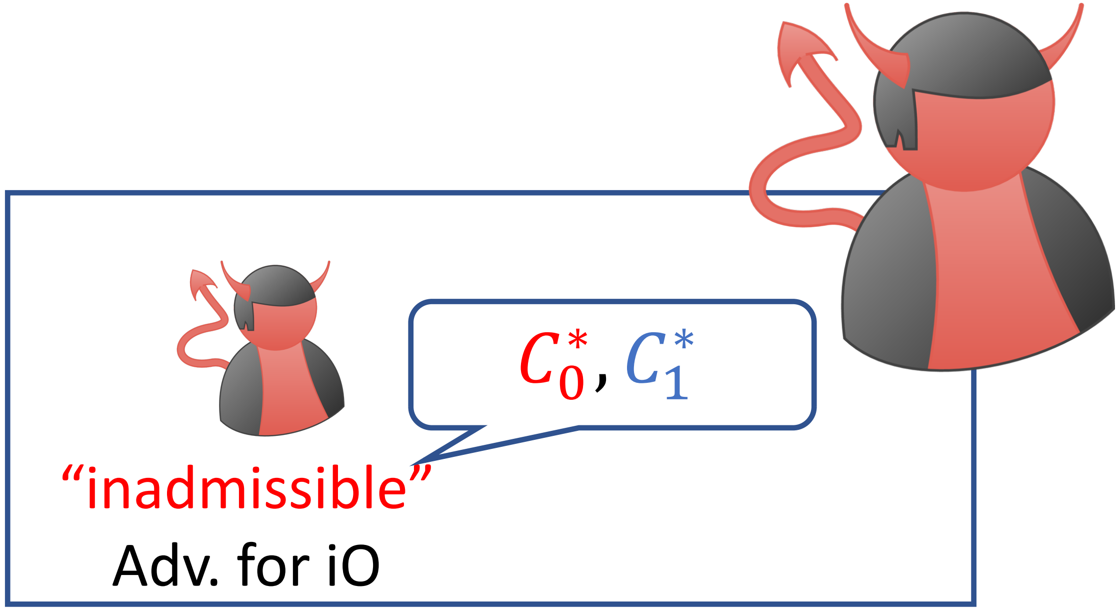
Is $2^{|input|}$-Loss Inherent? (folklore)



Is $2^{|input|}$-Loss Inherent? (folklore)

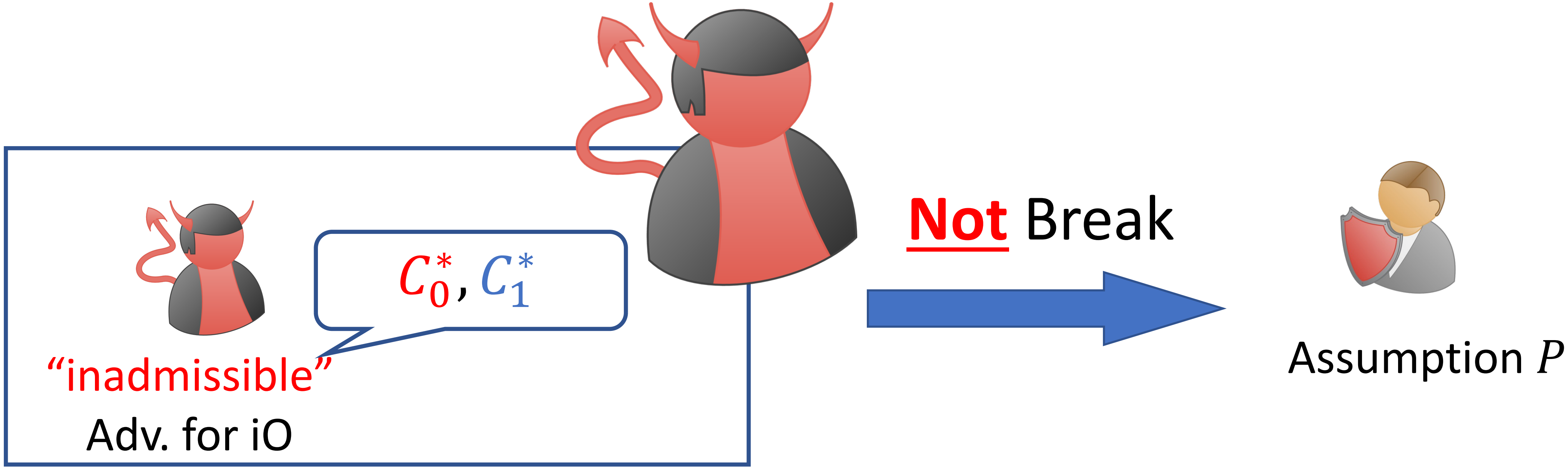


Is $2^{|input|}$-Loss Inherent? (folklore)



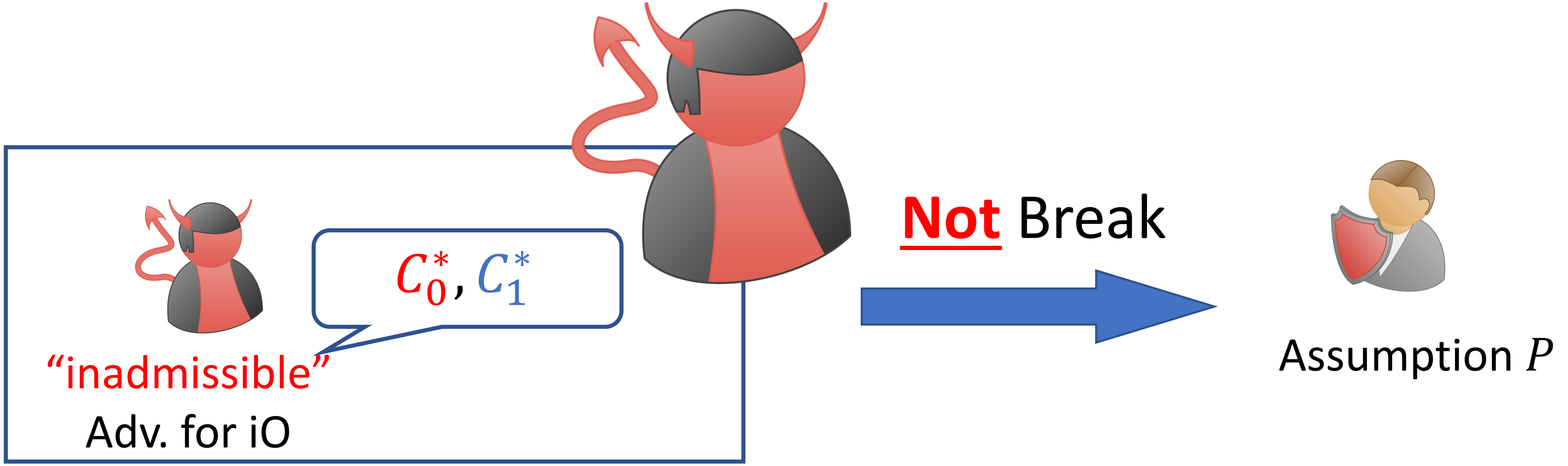
$C_0^*(x^*) \neq C_1(x^*)$

Is $2^{|input|}$-Loss Inherent? (folklore)



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

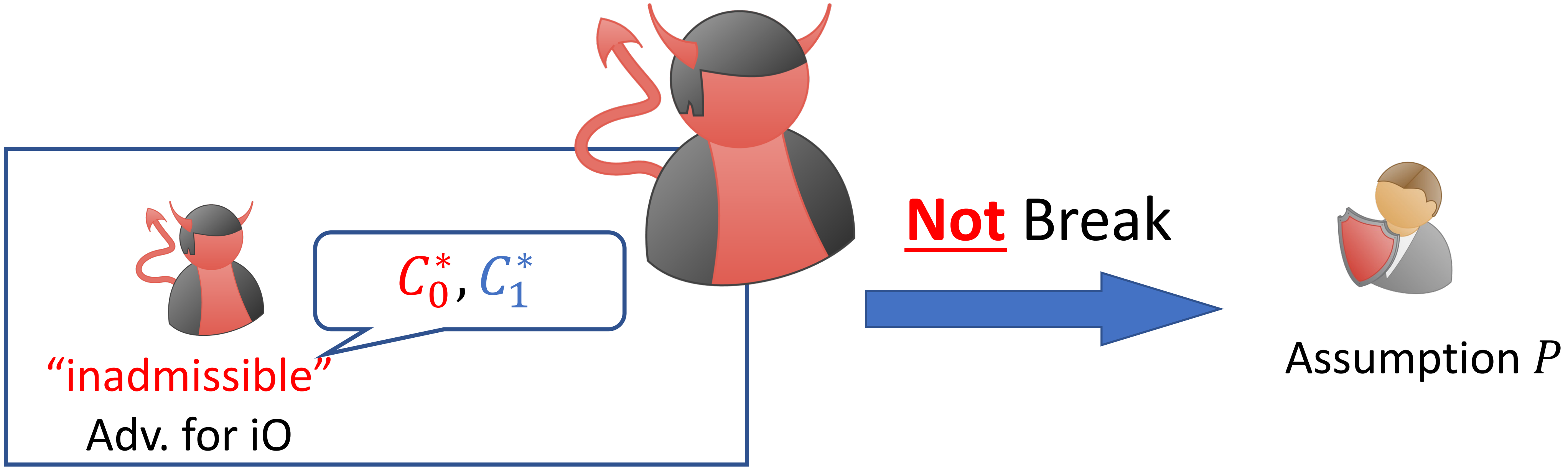
Is $2^{|input|}$-Loss Inherent? (folklore)



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

Reduction needs to check $C_0^* \equiv C_1^*$

Is $2^{|input|}$-Loss Inherent? (folklore)



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

Reduction needs to check $C_0^* \equiv C_1^*$

Broader Perspective

Broader Perspective

Non-Falsifiable definition appears in many other places in crypto

Broader Perspective

Non-Falsifiable definition appears in many other places in crypto

Example: Soundness of Proof Systems

(for $L \in NP$)

Broader Perspective

Non-Falsifiable definition appears in many other places in crypto

Example: Soundness of Proof Systems

(for $L \in NP$)

If $x \notin L$, any cheating proof should be rejected

Broader Perspective

Non-Falsifiable definition appears in many other places in crypto

Example: Soundness of Proof Systems

(for $L \in NP$)

$\boxed{\text{If } x \notin L,}$ any cheating proof should be rejected

Non-Falsifiable

Broader Perspective

Non-Falsifiable definition appears in many other places in crypto

Example: Soundness of Proof Systems

(for $L \in NP$)

$\boxed{\text{If } x \notin L,}$ any cheating proof should be rejected

Non-Falsifiable

[Gentry-Wichs'10] impossibility for SNARGs

iO

iO

Reduction checks $C_0(x^*) = C_1(x^*)$ for every x^* with $2^{|x^*|}$-loss

Previous Works:

iO

Reduction checks $C_0(x^*) = C_1(x^*)$ for every x^* with $2^{|x^*|}$-loss

Previous Works:

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

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

[Liu-Zhandry'17]

iO

Reduction checks $C_0(x^*) = C_1(x^*)$ for every x^* with $2^{|x^*|}$-loss

Previous Works:

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

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

[Liu-Zhandry'17]

This Work:

Leverage **math. proofs** of "$\forall x\ C_0(x) = C_1(x)$" to avoid the $2^{|x|}$-loss

Why Math. Proofs Exist?

Recall: when iO is used in the security proof

…

- Construct C_0, C_1
- **Write a math. proof** for $\forall\, x\ C_0(x) = C_1(x)$
- Apply iO security to derive $iO(C_0) \approx_c iO(C_1)$

…

The proof must be "**short**" (length $\ll 2^{|x|}$)

Otherwise, we (human brain) can't understand it.

Our Results I (for Propositional Logic)

iO with security loss independent of $|input|$ for any ckts $\{C_n^1\}_n, \{C_n^2\}_n$ where $C_n^1(x) \leftrightarrow C_n^2(x)$ have **poly-size proofs** in *Extended Frege systems.*

Our Results I (for Propositional Logic)

iO with security loss independent of $|input|$ for any ckts $\{C_n^1\}_n, \{C_n^2\}_n$ where $C_n^1(x) \leftrightarrow C_n^2(x)$ have **poly-size proofs** in *Extended Frege systems.*

(Assumptions: $2^{p(\lambda)}$-secure LWE, OWF,
iO for circuits of size independent of $|input|$.)

Extended Frege System ($\mathcal{EF}$)

Variables represent True/False

- **Axioms**:

$$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$$
$$p \to (q \to p)$$
$$p \to \neg\neg p$$

- **Inference Rule:**

$$p, p \to q \vdash q$$

- **Extension Rule:**

$$e \leftrightarrow \phi$$

(assign a new variable e to a formula ϕ)

What theorems have poly-size $\mathcal{EF}$-proofs?

Cook's Theory PV [Cook'75]

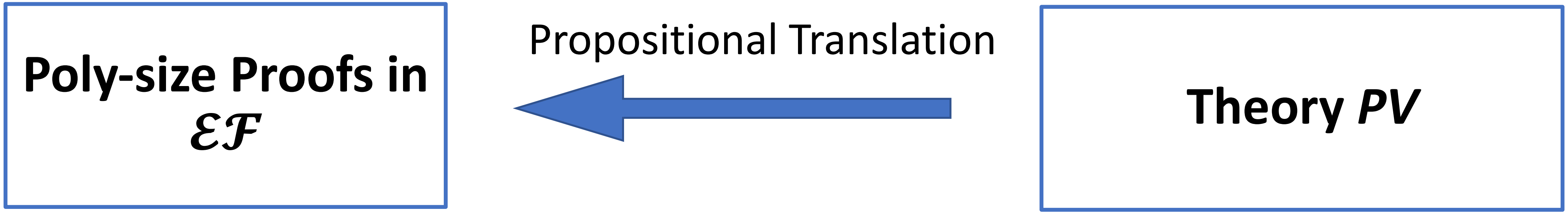
Cook's Theory PV [Cook'75]

Poly-size Proofs in $\mathcal{EF}$

Propositional Translation

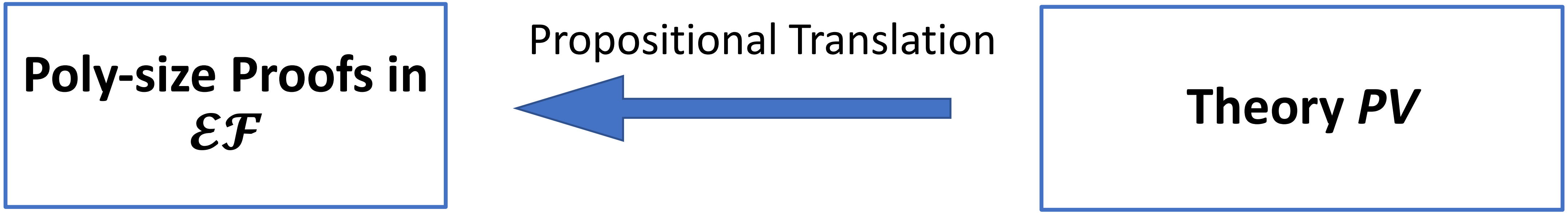
Theory *PV*

Cook's Theory PV [Cook'75]



Variables represent ***natural numbers***

Cook's Theory PV [Cook'75]



Variables represent ***natural numbers***

Allow definition of *any* polynomial-time functions, e.g.

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

Our Results II (for Cook's Theory PV)

iO for any ***unbounded-input*** Turing machines M_1, M_2,
with $\vdash_{PV} M_1(x) = M_2(x)$.

Assumptions: sub-exponential security of LWE & iO for circuits.

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 Thoerems (assuming Factoring):

- Fermat’s Little Theorem
- Correctness for “Primes is in P”

Our Results III: Applications

SNARGs with CRS size $\mathrm{poly}(\lambda, T_{\bar{R}})$ for $L \in NP \cap \mathrm{coNP}$, if

Our Results III: Applications

SNARGs with CRS size $\mathrm{poly}(\lambda, T_{\bar{R}})$ for $L \in NP \cap \mathrm{coNP}$, if

"$L \cap \bar{L} = \phi$" is provable in PV,

$$(\vdash_{PV} \bar{R}(x, \overline{w}) = 1 \rightarrow R(x, w) = 0)$$

R (resp. $\bar{R}$) is NP-relation machine of L (resp. $\bar{L}$)

Our Results III: Applications

SNARGs with CRS size $\mathrm{poly}(\lambda, T_{\bar{R}})$ for $L \in NP \cap \mathrm{coNP}$, if

"$L \cap \bar{L} = \phi$" is provable in PV,

$(\vdash_{PV} \bar{R}(x, \bar{w}) = 1 \rightarrow R(x, w) = 0)$

R (resp. $\bar{R}$) is NP-relation machine of L (resp. $\bar{L}$)

(Also apply to witness encryptions with ciphertext size $\mathrm{poly}(\lambda, T_{\bar{R}})$)

How do we leverage math. proofs?

How do we leverage math. proofs?

(An Overview)

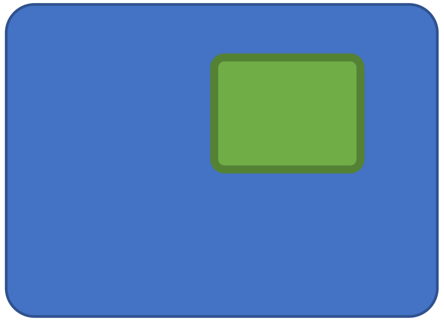
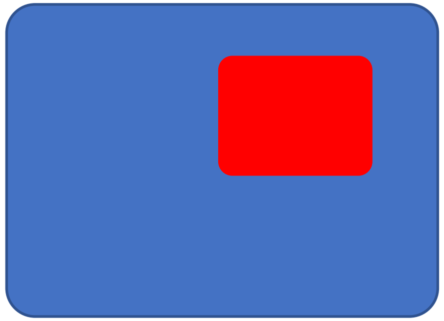
δ-Equivalence for Circuits

δ-Equivalence for Circuits

C:

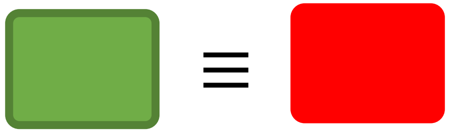
: C'

δ-Equivalence for Circuits

C:   : C'

C and C' are almost the same, except for

a **functionality equivalent *<u>sub-circuit</u>*** of size $O(\log n)$

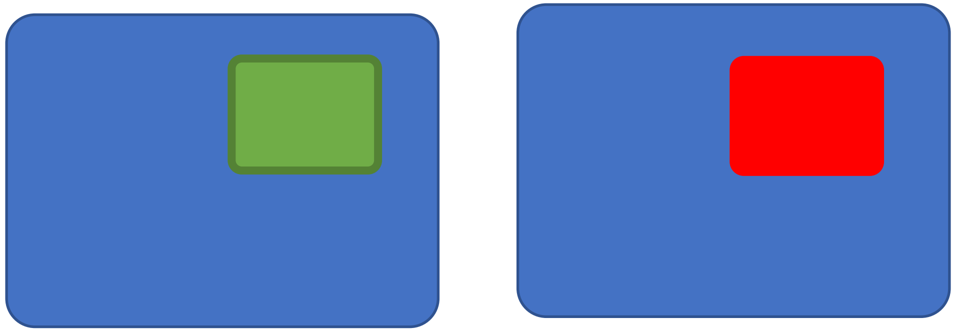


$\mathcal{EF}$-Proofs imply δ-Equivalence

Poly. $\mathcal{EF}$ proof for $C_0(x) \leftrightarrow C_1(x)$

$C_0 \equiv C^{(1)}$ $\quad$ $C^{(2)}$ $\quad$ $C^{(3)}$ $\quad$ … $\quad$ $C^{(poly)} \equiv C_1$

…



…

$C^{(i)}$ and $C^{(i+1)}$ are δ-equivalent

Focus: iO for δ-Equivalent Ckts

Focus: iO for δ-Equivalent Ckts

Assume iO for δ-Equivalent Ckts: δiO

Focus: iO for δ-Equivalent Ckts

Assume iO for δ-Equivalent Ckts: δiO

$\delta iO(C^{(1)})$ $\delta iO(C^{(2)})$ $\delta iO(C^{(3)})$ … $\delta iO(C^{(\ell')})$

Focus: iO for δ-Equivalent Ckts

Assume iO for δ-Equivalent Ckts: δiO

$$\delta iO(C^{(1)}) \quad \approx \quad \delta iO(C^{(2)}) \quad \approx \quad \delta iO(C^{(3)}) \quad \dots \quad \delta iO(C^{(\ell')})$$

Focus: iO for δ-Equivalent Ckts

Assume iO for δ-Equivalent Ckts: δiO

$$\delta iO(C^{(1)}) \quad \approx \quad \delta iO(C^{(2)}) \quad \approx \quad \delta iO(C^{(3)}) \quad \ldots \quad \delta iO(C^{(\ell')})$$

$$\Rightarrow \delta iO(C_0) \approx_c \delta iO(C_1)$$

Focus: iO for δ-Equivalent Ckts

Assume iO for δ-Equivalent Ckts: δiO

$$\delta iO(C^{(1)}) \quad \approx \quad \delta iO(C^{(2)}) \quad \approx \quad \delta iO(C^{(3)}) \quad \ldots \quad \delta iO(C^{(\ell')})$$

$$\Rightarrow \delta iO(C_0) \approx_c \delta iO(C_1)$$

<u>**Total Security Loss**</u> **=** ℓ' $\cdot$Loss of δiO $(\ell' = poly)$

Focus: iO for δ-Equivalent Ckts

Assume iO for δ-Equivalent Ckts: δiO

$$\delta iO(C^{(1)}) \quad \approx \quad \delta iO(C^{(2)}) \quad \approx \quad \delta iO(C^{(3)}) \quad \ldots \quad \delta iO(C^{(\ell')})$$

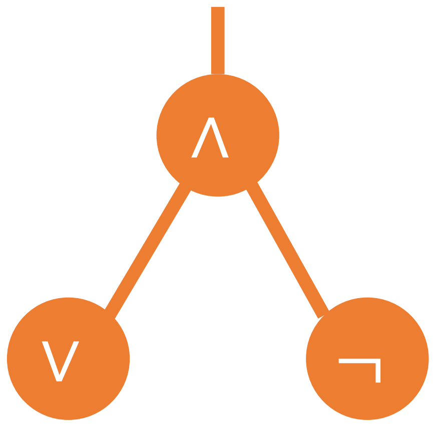
$$\Rightarrow \delta iO(C_0) \approx_c \delta iO(C_1)$$

Total Security Loss = ℓ' $\cdot$Loss of δiO $\quad (\ell' = poly)$

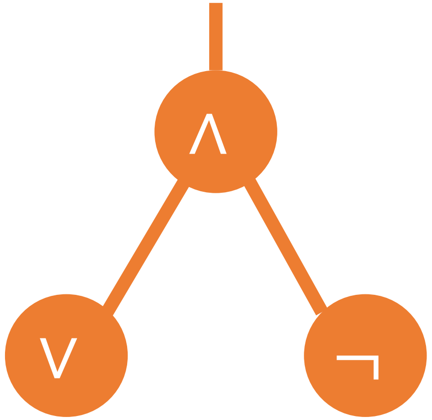
If loss of δiO is independent of $|input|$, so is the total loss.

Constructing δiO

Constructing δiO



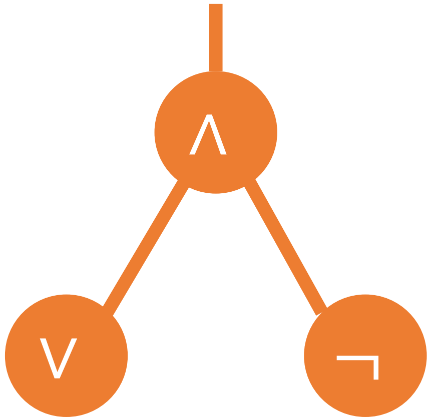
Constructing δiO



"Gate-by-Gate" Obfuscation:

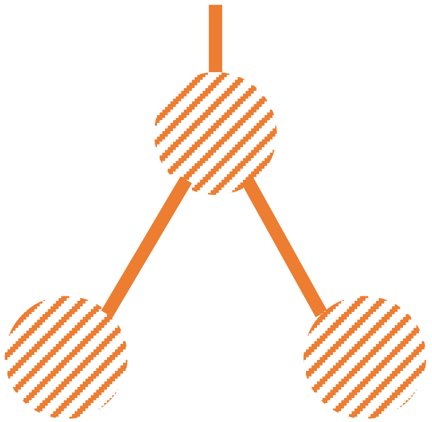
Obfuscate each gate separately

Constructing δiO

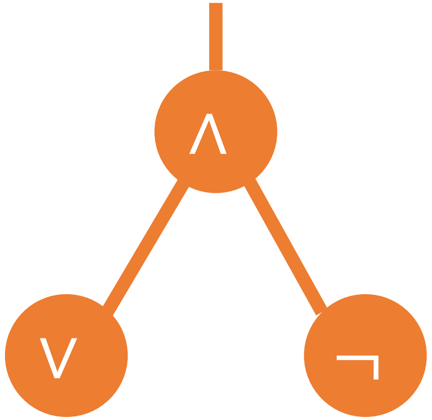


"Gate-by-Gate" Obfuscation:

Obfuscate each gate separately

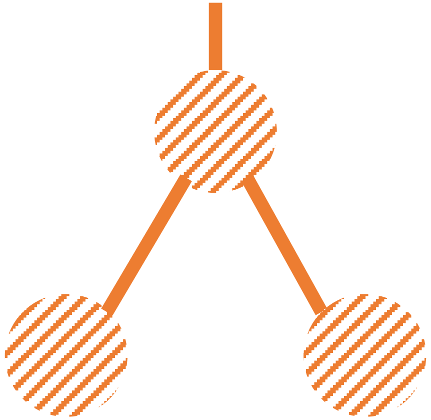


Constructing δiO



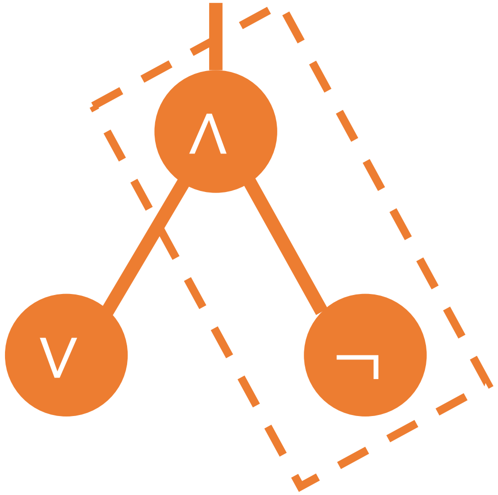
"Gate-by-Gate" Obfuscation:

Obfuscate each gate separately



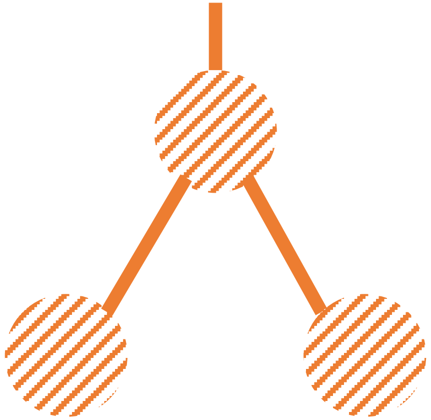
Topology is preserved

Constructing δiO



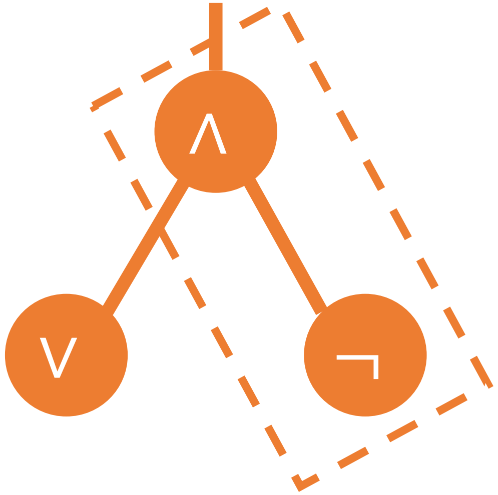
"Gate-by-Gate" Obfuscation:

Obfuscate each gate separately



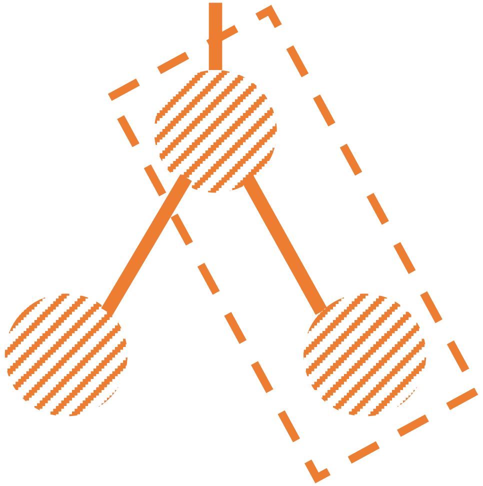
Topology is preserved

Constructing δiO



"Gate-by-Gate" Obfuscation:

Obfuscate each gate separately

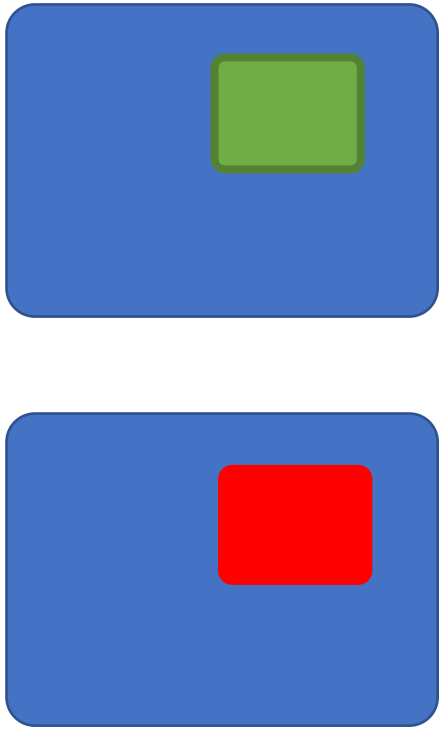


Topology is preserved

Security Proof w/o $2^{|input|}$-Loss

Security Proof w/o $2^{|input|}$-Loss

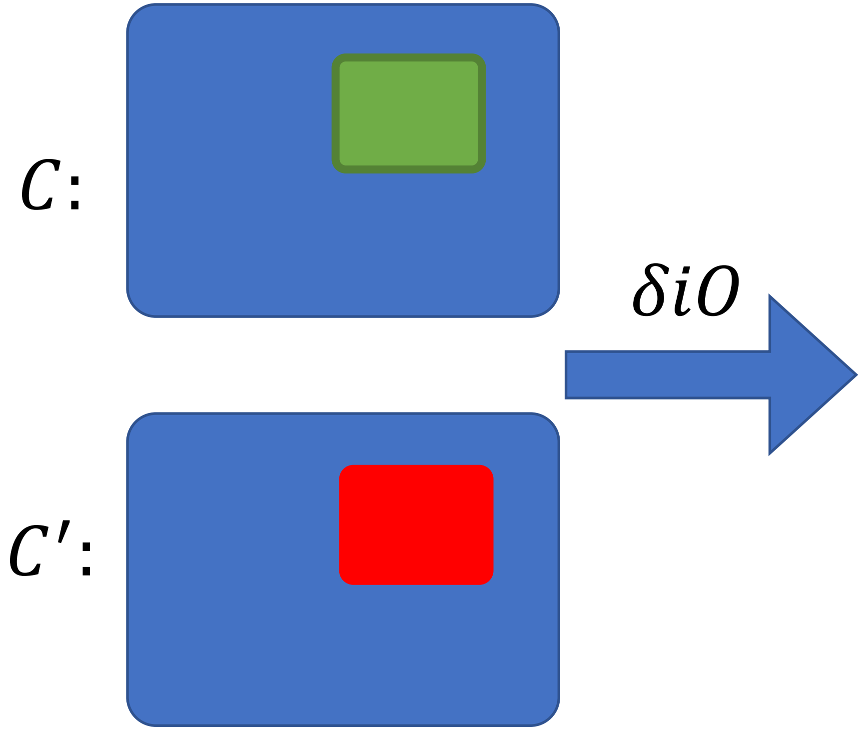
C:



C':

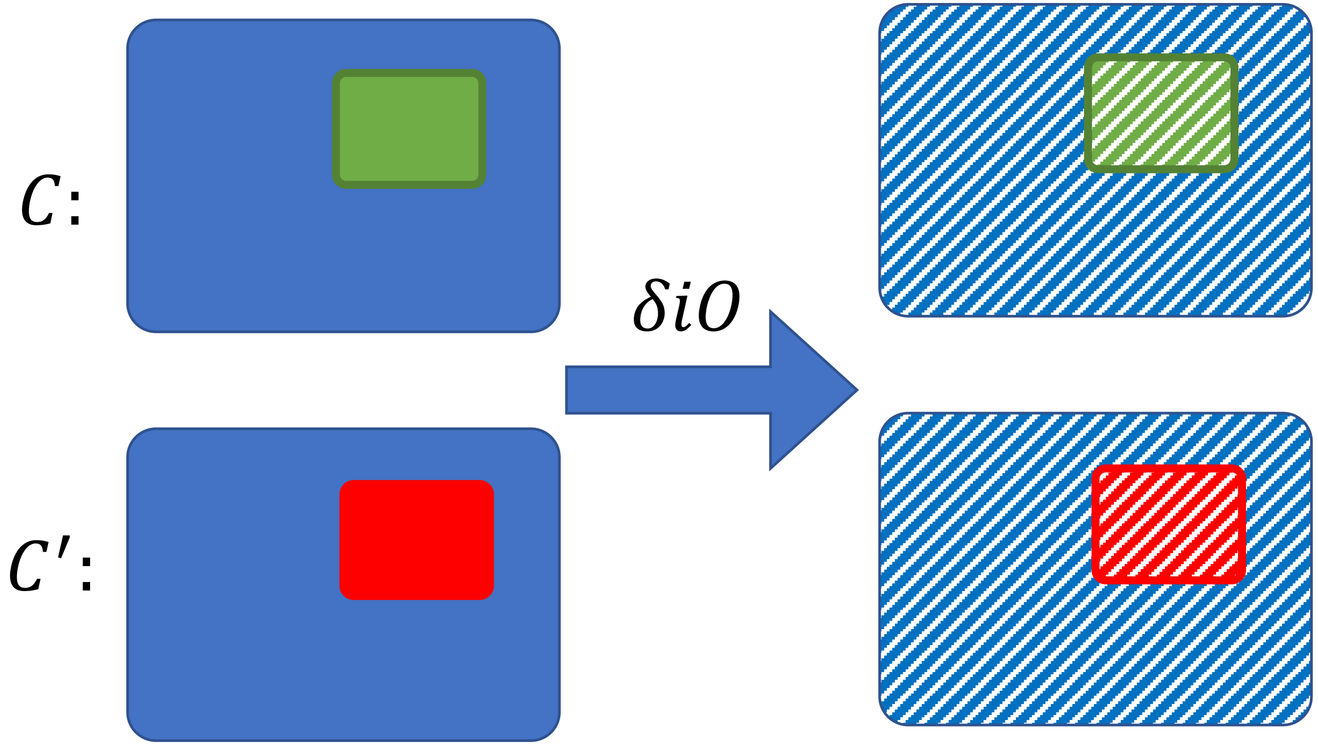
C, C': δ-Equivalent

Security Proof w/o $2^{|input|}$-Loss



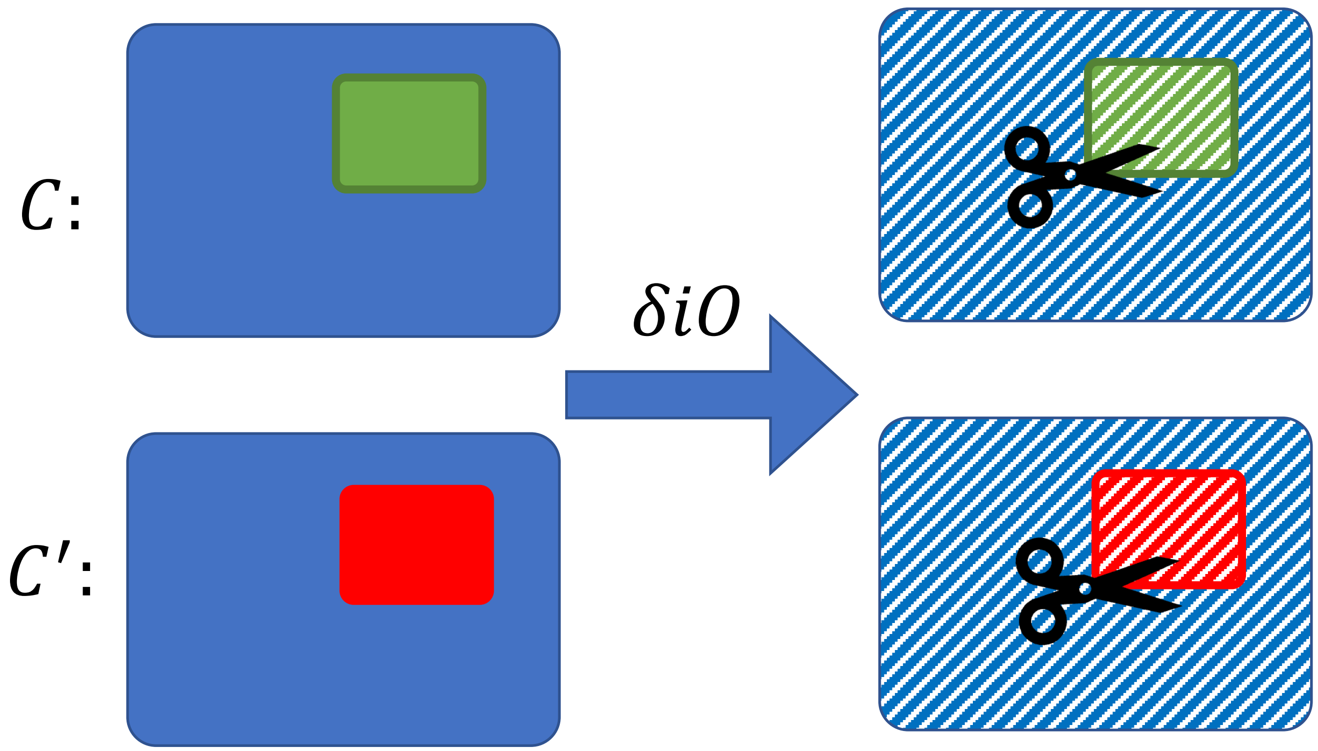
C, C': δ-Equivalent

Security Proof w/o $2^{|input|}$-Loss



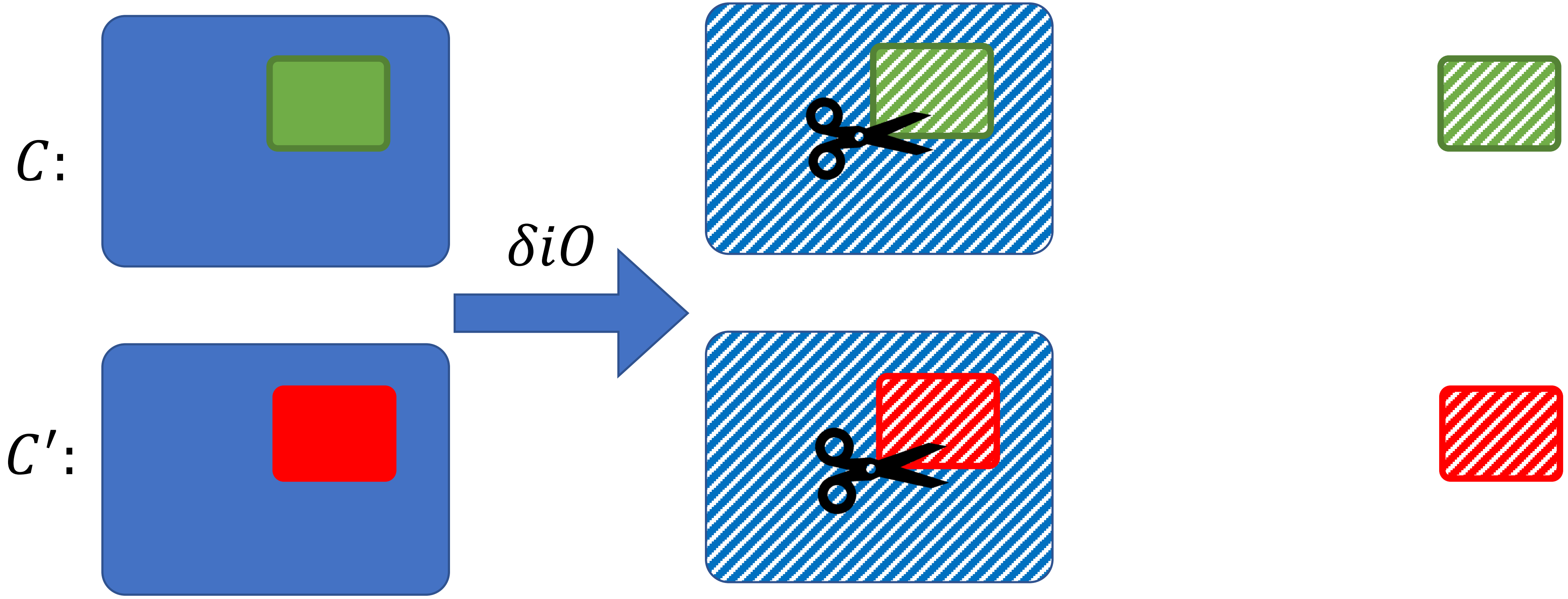
C, C': δ-Equivalent

Security Proof w/o $2^{|input|}$-Loss



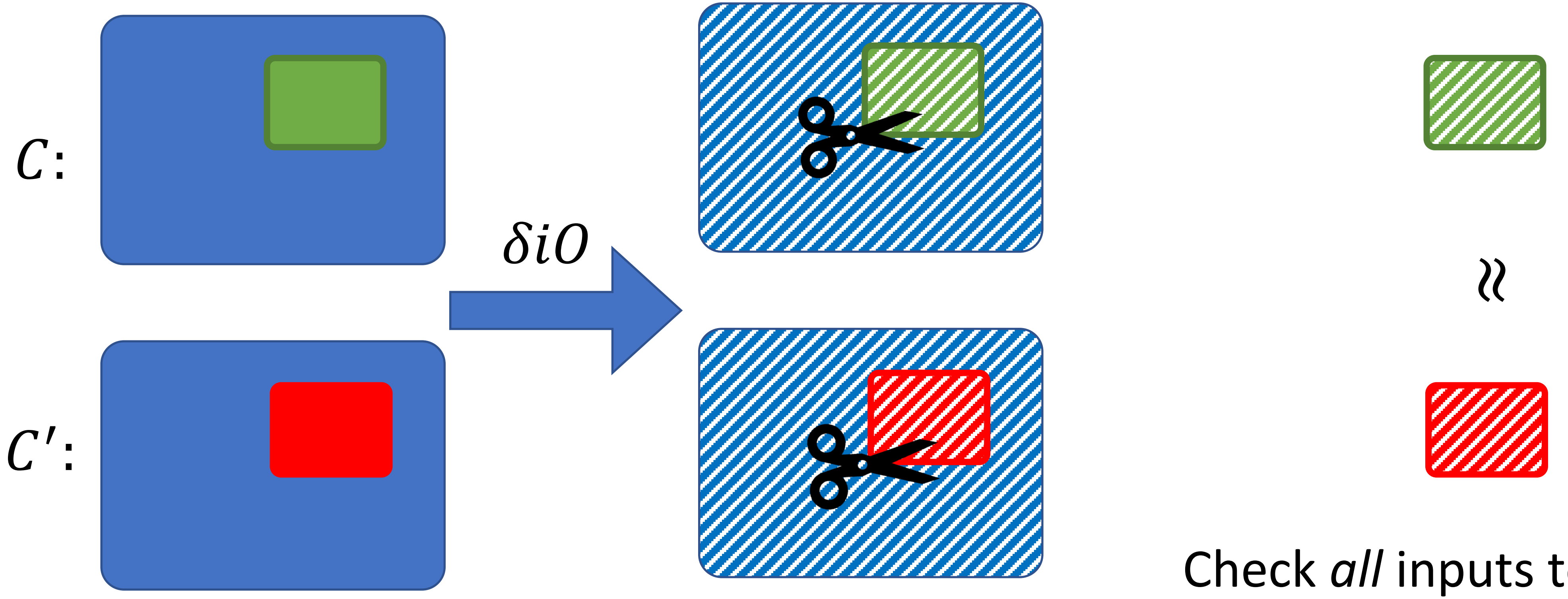
C, C': δ-Equivalent

Security Proof w/o $2^{|input|}$-Loss



C, C': δ-Equivalent

Security Proof w/o $2^{|input|}$-Loss



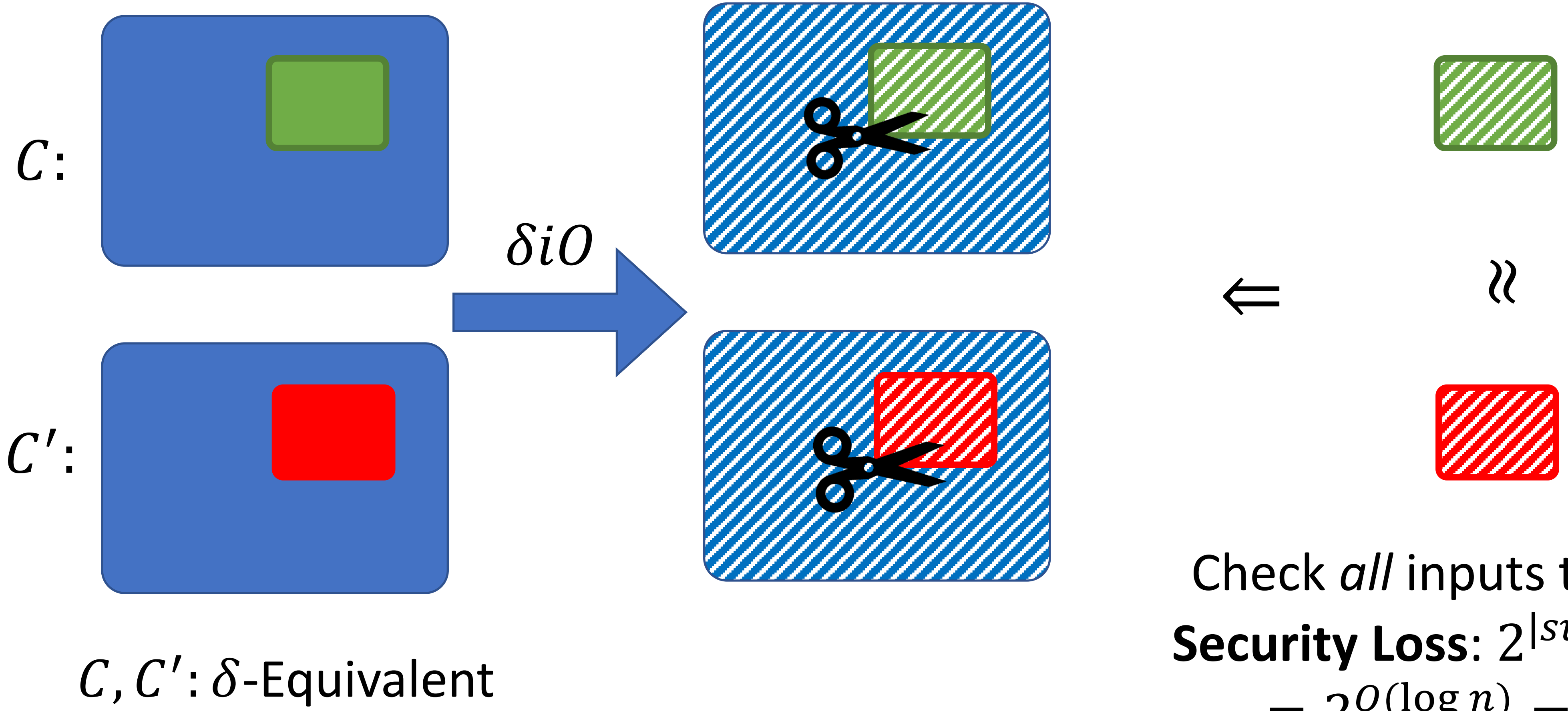
C, C': δ-Equivalent

Check *all* inputs to **Sub-ckt**

Security Loss: $2^{|subckt\ input|}$

$= 2^{O(\log n)} = poly$!

Security Proof w/o $2^{|input|}$-Loss

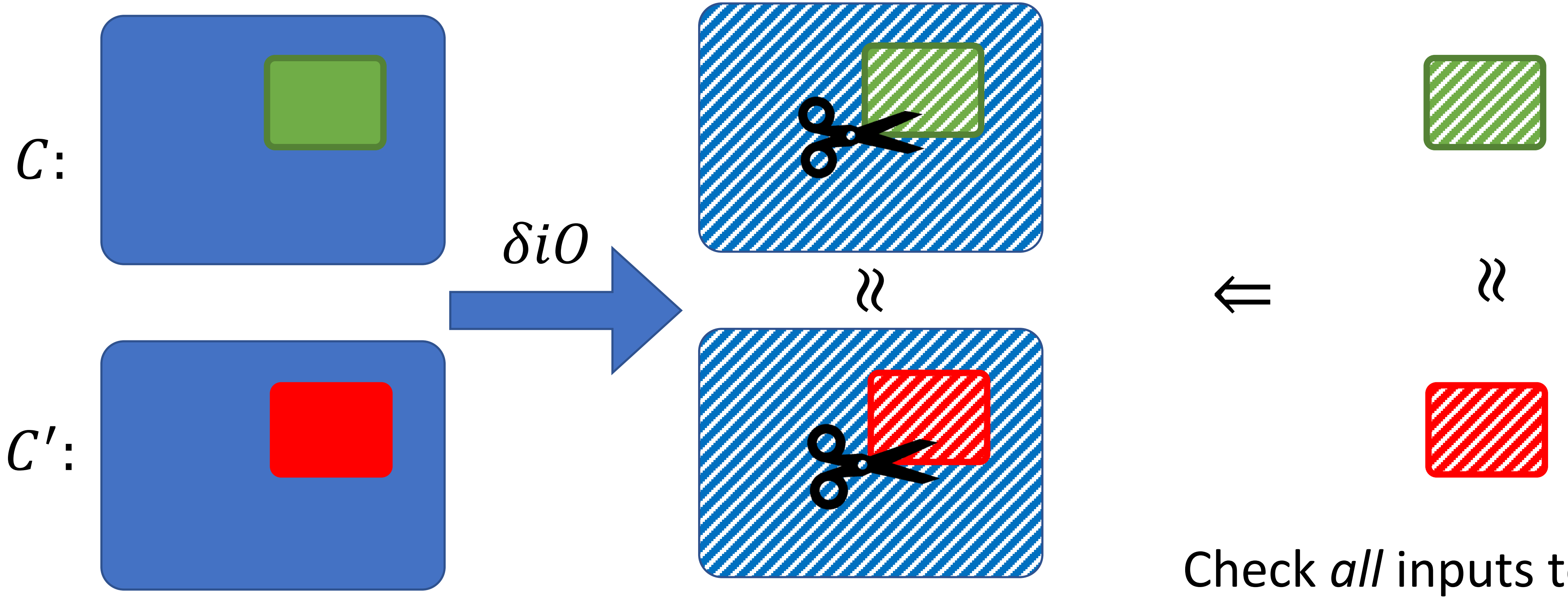


Check *all* inputs to **Sub-ckt**

Security Loss: $2^{|subckt\ input|}$

$= 2^{O(\log n)} = poly$!

Security Proof w/o $2^{|input|}$-Loss



C, C': δ-Equivalent

Check *all* inputs to **Sub-ckt**

Security Loss: $2^{|subckt\ input|}$

$= 2^{O(\log n)} = poly$!

Technical Details

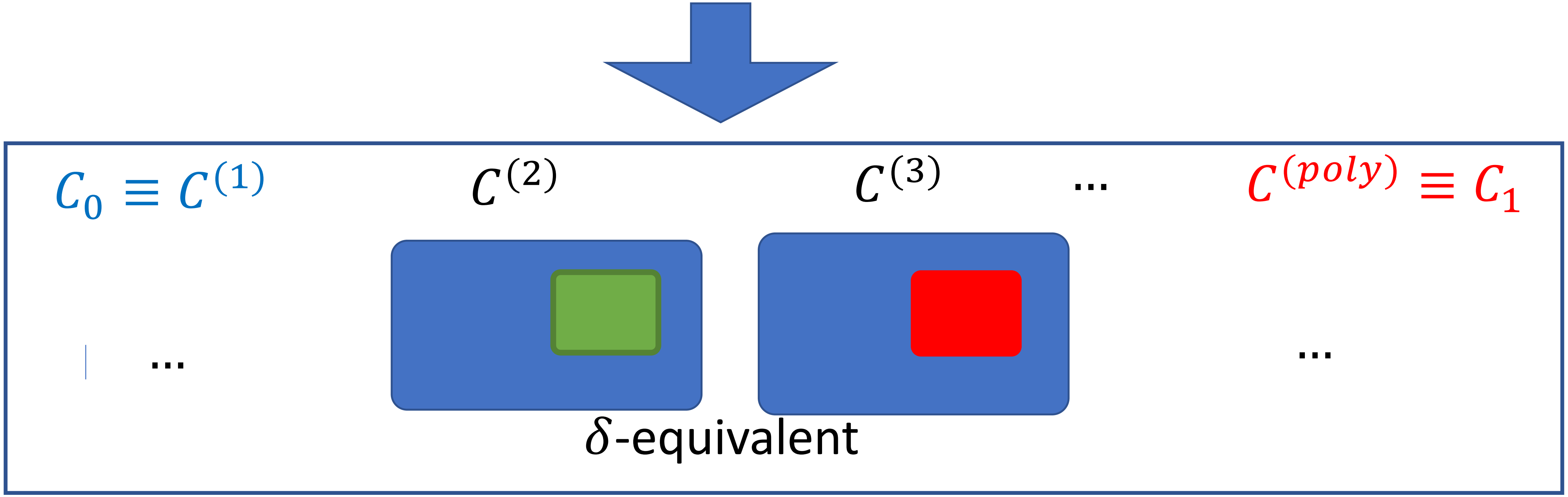
- $\mathcal{EF}$-Proofs $\Rightarrow$ δ-Equivalence
- Construct δiO
- iO for Turing machines

Technical Details

- **$\mathcal{EF}$-Proofs $\Rightarrow$ $\boldsymbol{\delta}$-Equivalence**
- Construct δiO
- iO for Turing machines

Goal: $\mathcal{EF}$-Proofs $\Rightarrow$ δ-Equivalence

Poly. $\mathcal{EF}$ proof for $C_0(x) \leftrightarrow C_1(x)$

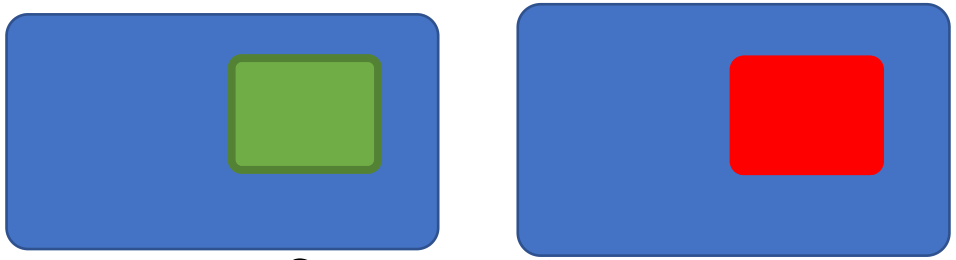


Goal: $\mathcal{EF}$-Proofs $\Rightarrow$ δ-Equivalence

Poly. $\mathcal{EF}$ proof for $C_0(x) \leftrightarrow C_1(x)$

$C_0 \equiv C^{(1)}$ $\quad C^{(2)} \quad C^{(3)} \quad \ldots \quad C^{(poly)} \equiv C_1$

…



…

δ-equivalent

Alternative View: A sequence of ***local*** changes

Key Observation

Key Observation

Proofs in logic systems are “local”

Key Observation

Proofs in logic systems are “local”

(Similar to δ-equivalence)

Key Observation

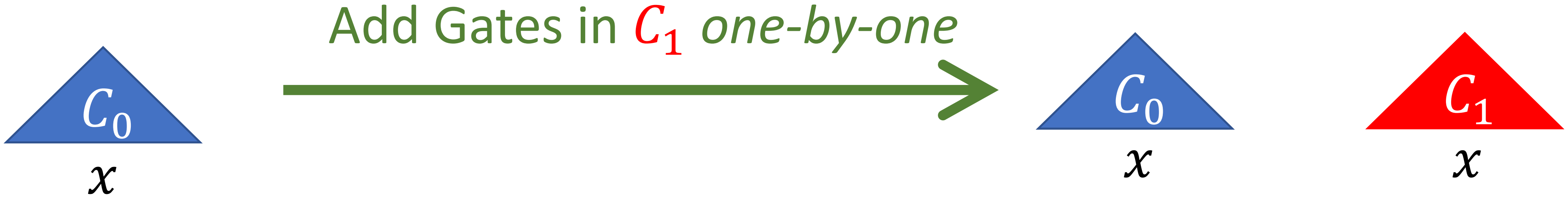
Proofs in logic systems are "local"

(Similar to δ-equivalence)

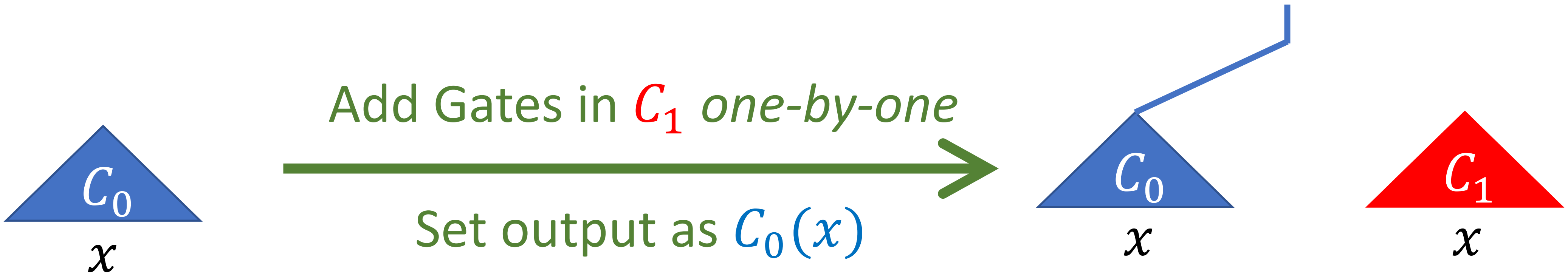
Each line in $\mathcal{EF}$-proofs is also a circuit

(Can be used to modify circuits)

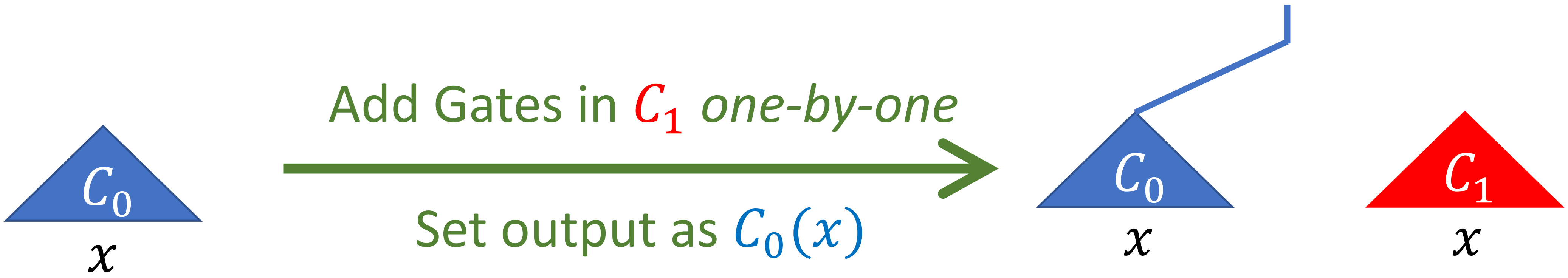
Stage I: Grow C_1



Stage I: Grow C_1



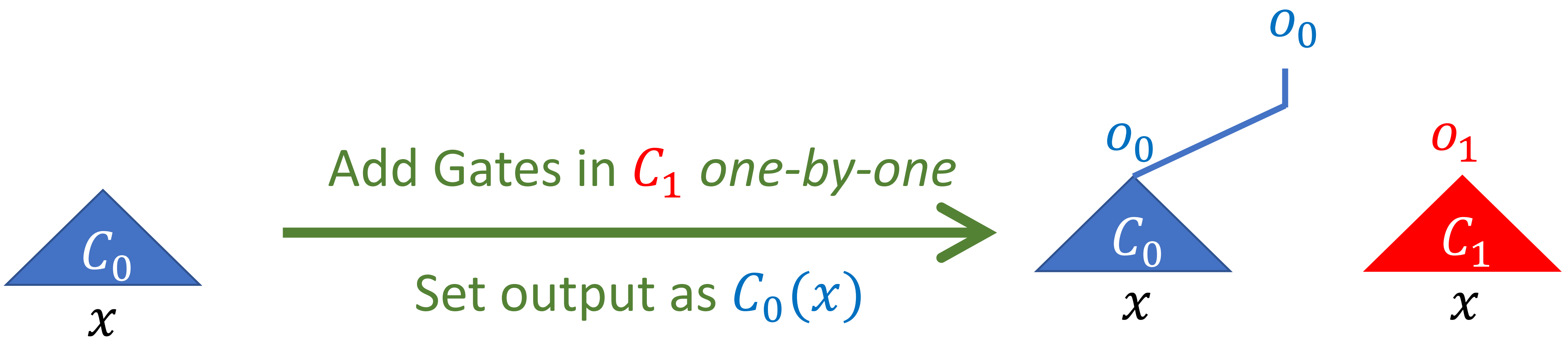
Stage I: Grow C_1



δ-Equivalence

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

Stage I: Grow C_1



<u>δ-Equivalence</u>

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

Stage II: Grow the Proof

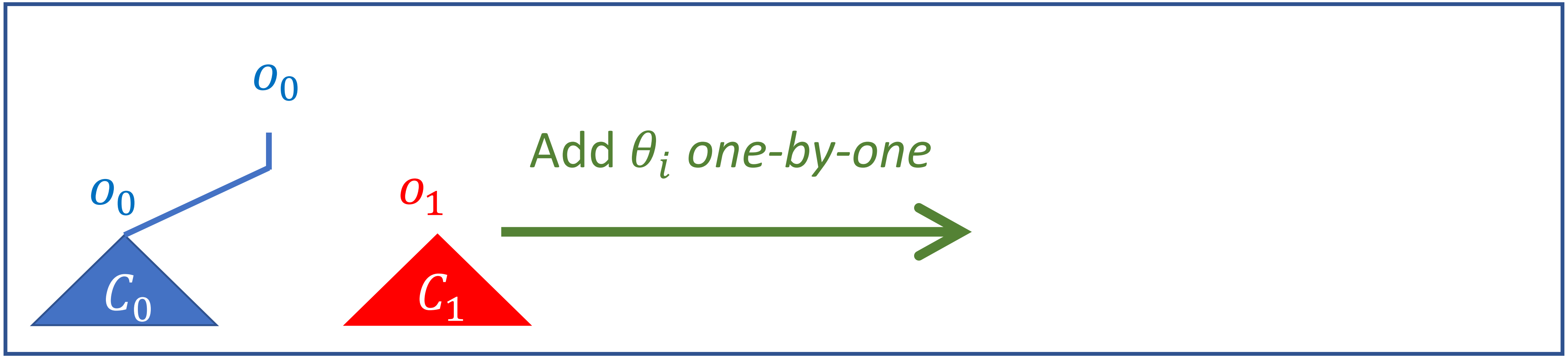
$\mathcal{EF}$-Proof of $C_0(x) \leftrightarrow C_1(x)$: $\theta_1, \theta_2, \ldots, \theta_\ell$

Stage II: Grow the Proof

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

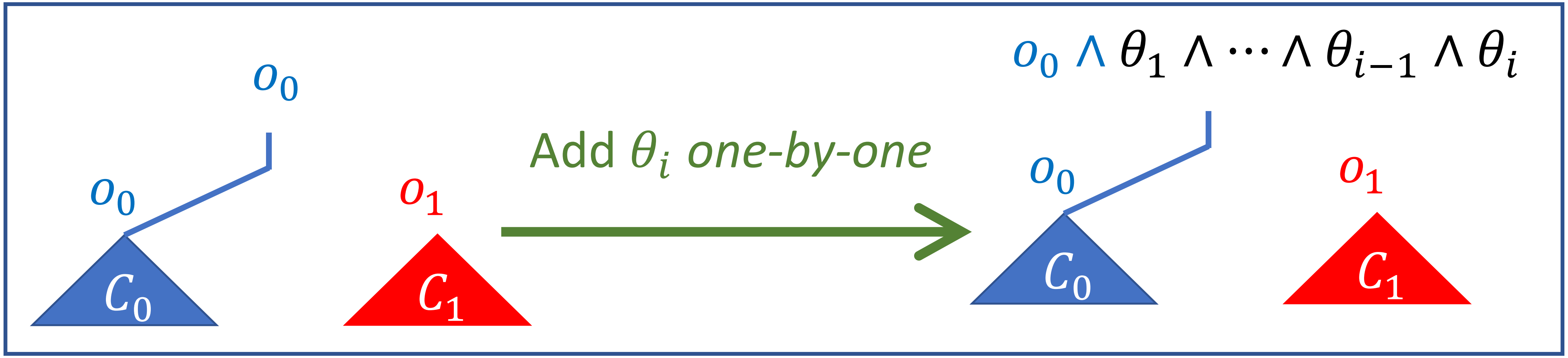
Stage II: Grow the Proof

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



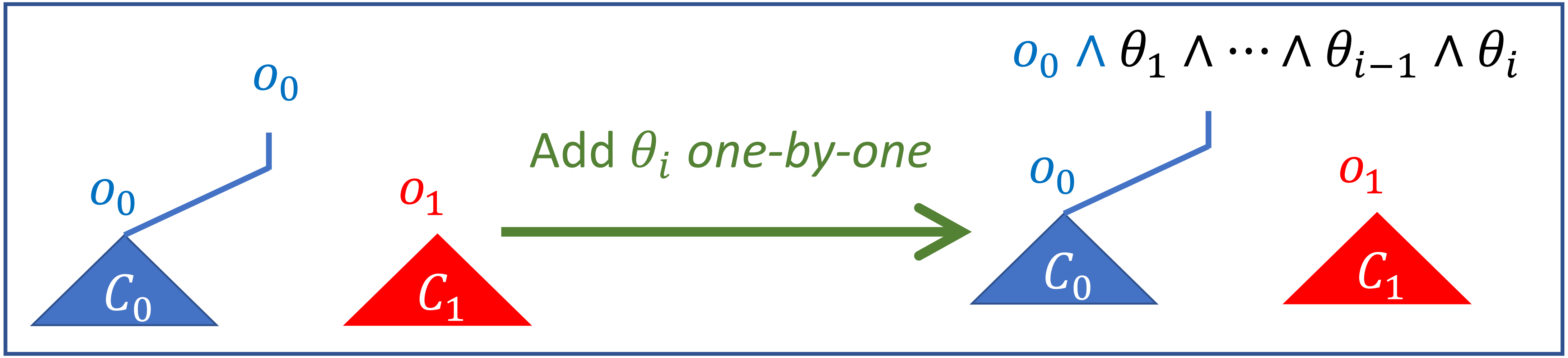
Stage II: Grow the Proof

$\mathcal{EF}$-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: δ-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: δ-Equivalence

i-th Step: Add θ_i

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

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

How θ_i is derived:

Stage II: δ-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> $C_0(x) \wedge \theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_i$

<u>How θ_i is derived:</u>

- Axiom

Stage II: δ-Equivalence

<u>i-th Step: Add θ_i</u>

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

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

<u>How θ_i is derived:</u>

- Axiom

Stage II: δ-Equivalence

<u>i-th Step: Add θ_i</u>

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

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

<u>How θ_i is derived:</u>

- Axiom $1 \equiv \theta_i$ (Axioms are tautologies)

Stage II: δ-Equivalence

<ins>i-th Step: Add θ_i</ins>

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

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

<ins>How θ_i is derived:</ins>

- Axiom

Stage II: δ-Equivalence

<u>i-th Step: Add θ_i</u>

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

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

<u>How θ_i is derived:</u>

- Axiom
- Inference Rule: Modus Ponens $(p, p \to q \vdash q)$

Stage II: δ-Equivalence

<u>i-th Step: Add θ_i</u>

<u>Before:</u>

<u>After:</u>

<u>How θ_i is derived:</u>

- Axiom
- Inference Rule: Modus Ponens $(p, p \rightarrow q \vdash q)$

Stage II: δ-Equivalence

<u>i-th Step: Add θ_i</u>

<u>Before:</u> $C_0(x) \land p \land \cdots \land (p \to q) \land \cdots$

<u>After:</u> $C_0(x) \land p \land \cdots \land (p \to q) \land \cdots \land q$

<u>How θ_i is derived:</u>

- Axiom
- Inference Rule: Modus Ponens $(p, p \to q \vdash q)$

Stage II: δ-Equivalence

<u>i-th Step: Add θ_i</u>

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

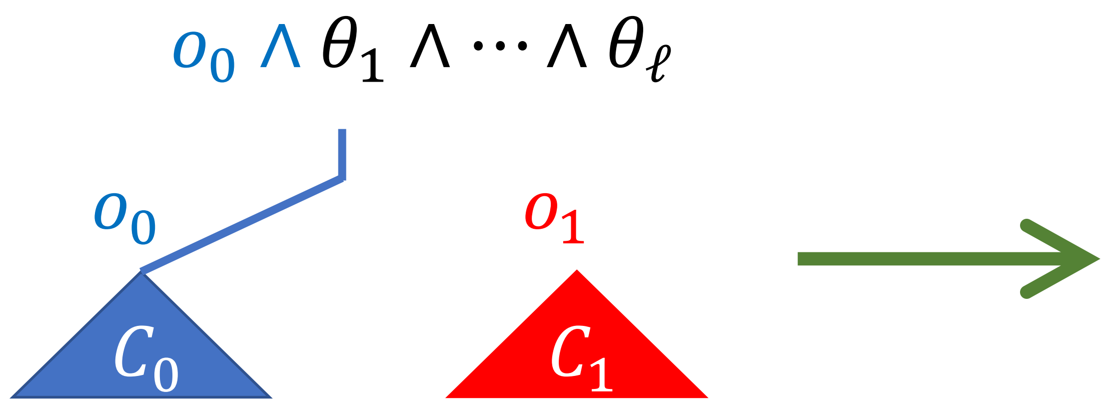
After: $C_0(x) \land p \land \cdots \land (p \to q) \land \cdots \land q$

<u>How θ_i is derived:</u>

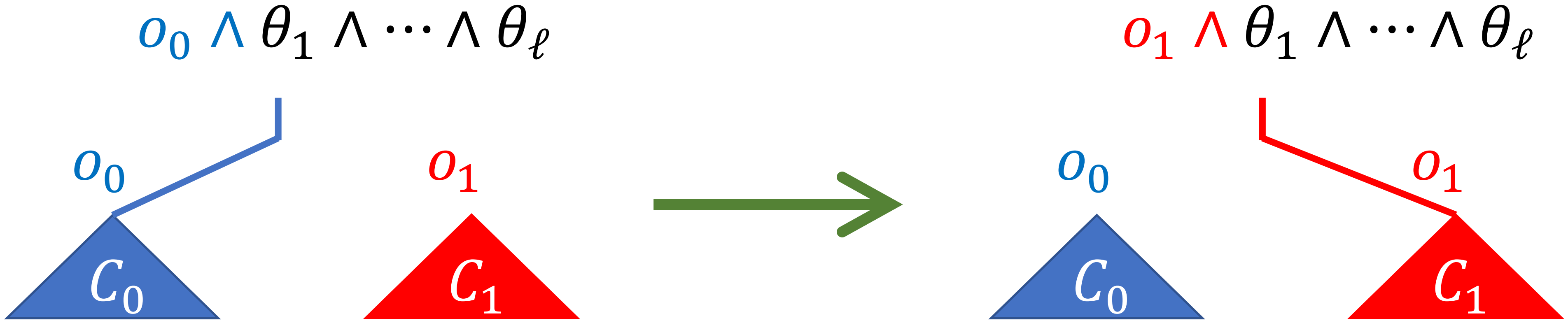
- Axiom
- Inference Rule: Modus Ponens $(p, p \to q \vdash q)$

$$p \land (p \to q) \equiv p \land (p \to q) \land q$$

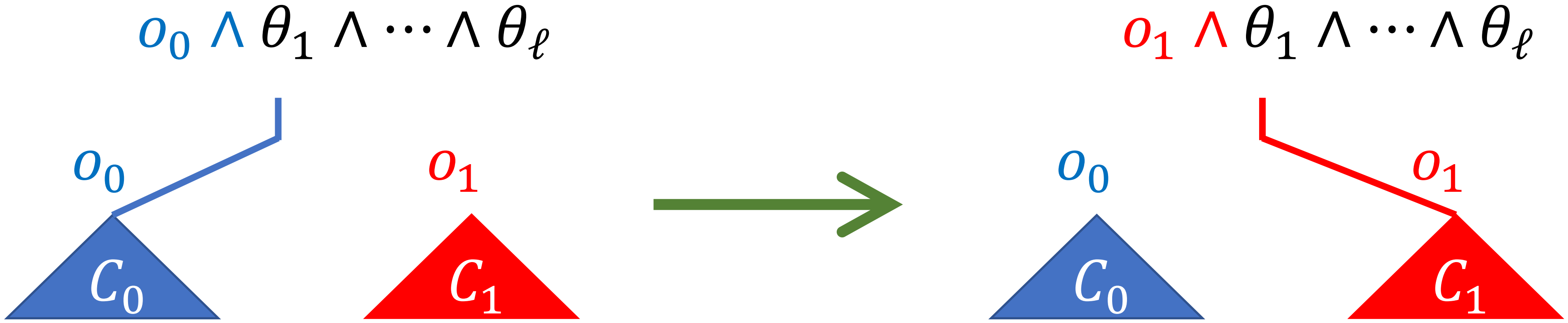
Stage III: **Switch** o_0 to o_1



Stage III: **Switch** o_0 to o_1



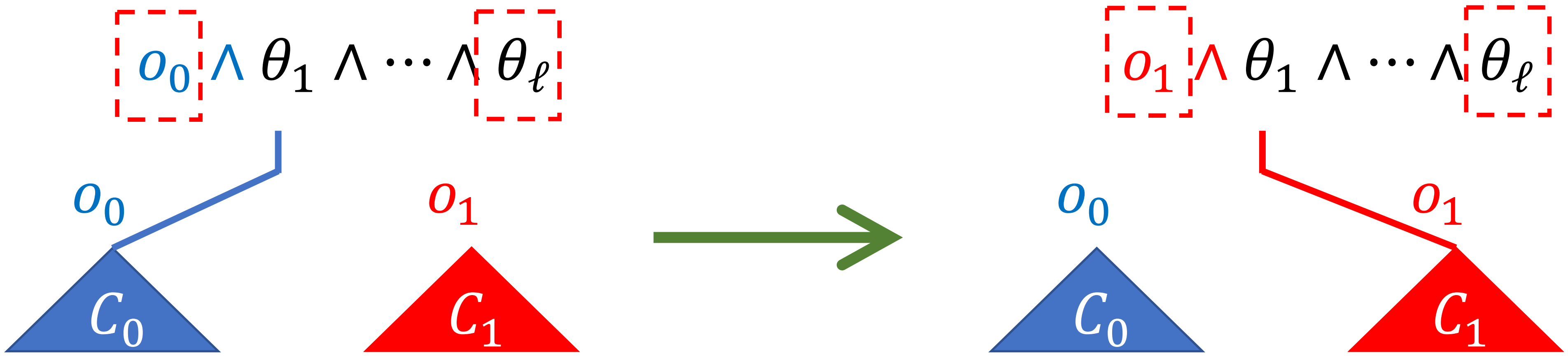
Stage III: **Switch** o_0 to o_1



δ-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



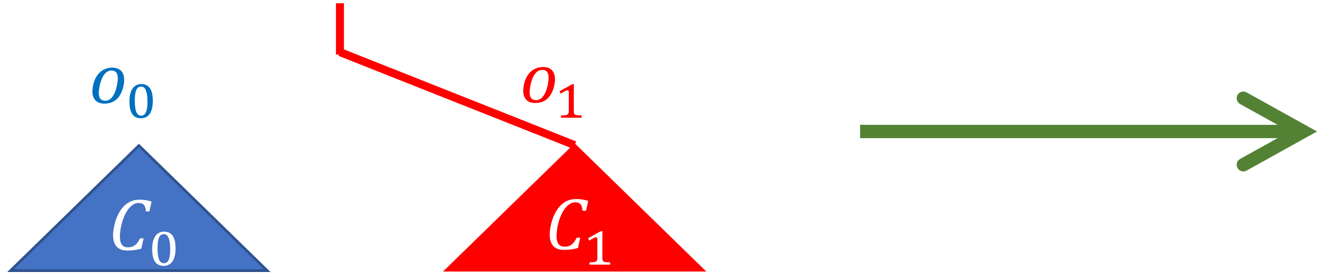
δ-Equivalence

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

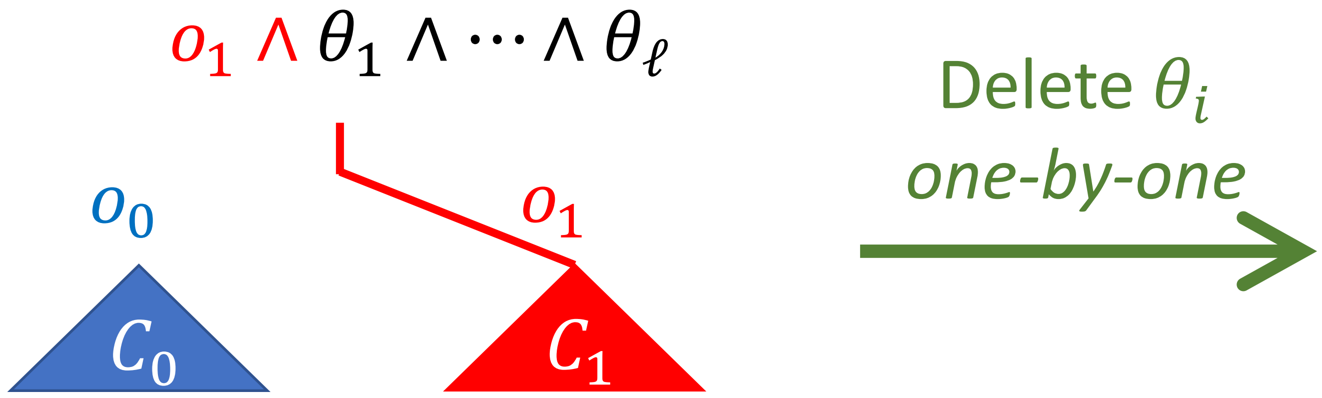
$$o_0 \wedge (o_0 \leftrightarrow o_1) \equiv o_1 \wedge (o_0 \leftrightarrow o_1)$$

Stage IV: Shrink the Proof

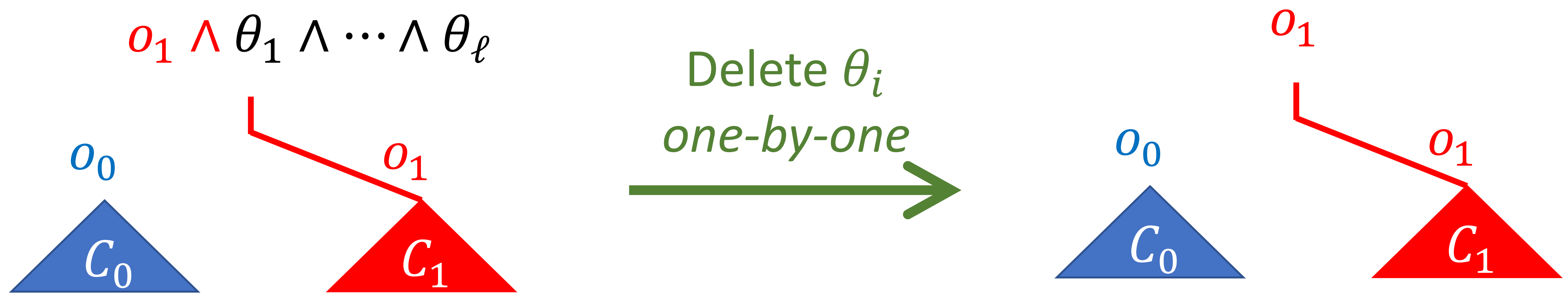
$o_1 \wedge \theta_1 \wedge \cdots \wedge \theta_\ell$



Stage IV: Shrink the Proof



Stage IV: Shrink the Proof

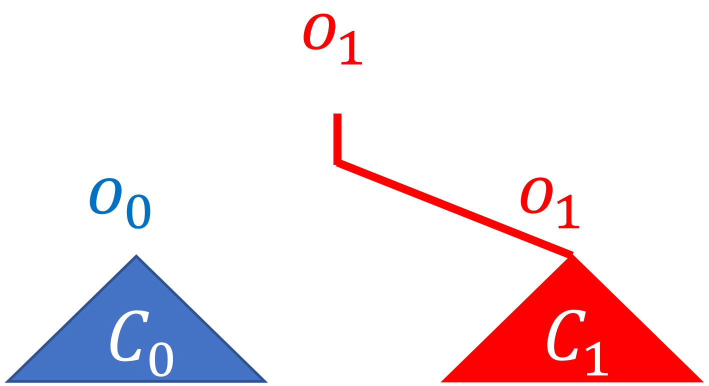


Stage IV: Shrink the Proof

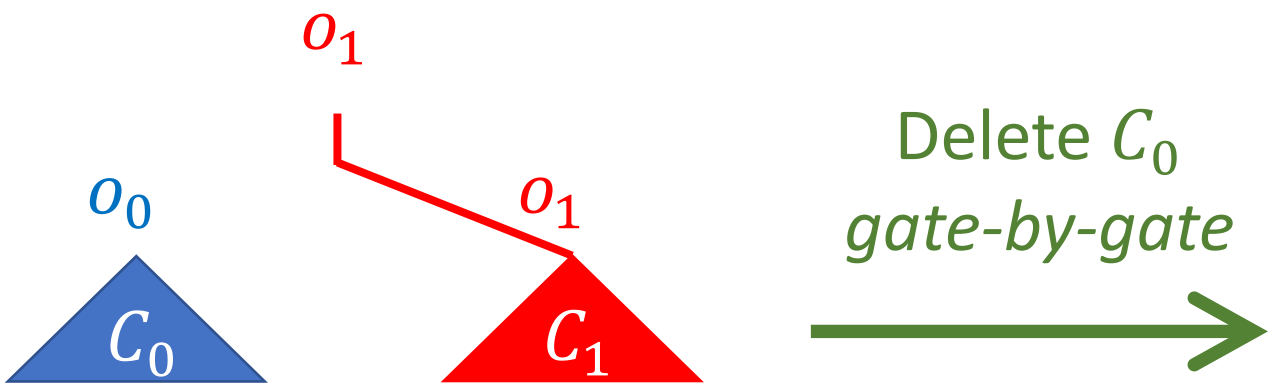
δ-Equivalence: Similar to "Growing the proof" Stage

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

δ-Equivalence:

Before we delete a gate,
the output of that gate is never used.

More Details: Multi-Arity Gates?

We Use: Multi-arity ∧-Gate

$$C_0(x) \wedge \theta_1 \wedge \theta_2 \ldots \wedge \theta_\ell$$

More Details: Multi-Arity Gates?

We Use: Multi-arity $\wedge$-Gate

$$C_0(x) \wedge \theta_1 \wedge \theta_2 \ldots \wedge \theta_\ell$$

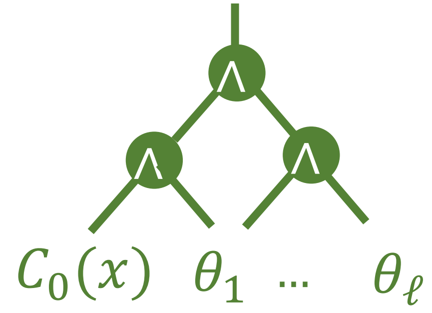
$\boldsymbol{\delta iO}$: Only Support $\boldsymbol{O}(\mathbf{1})$-arity Gates

More Details: Multi-Arity Gates?

We Use: Multi-arity ∧-Gate

$$C_0(x) \wedge \theta_1 \wedge \theta_2 \ldots \wedge \theta_\ell$$

Arity-2 ∧-Tree

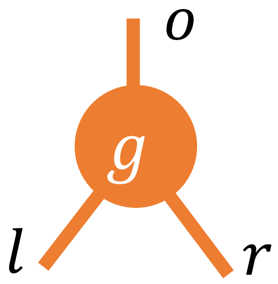


δiO: Only Support $O(1)$-arity Gates

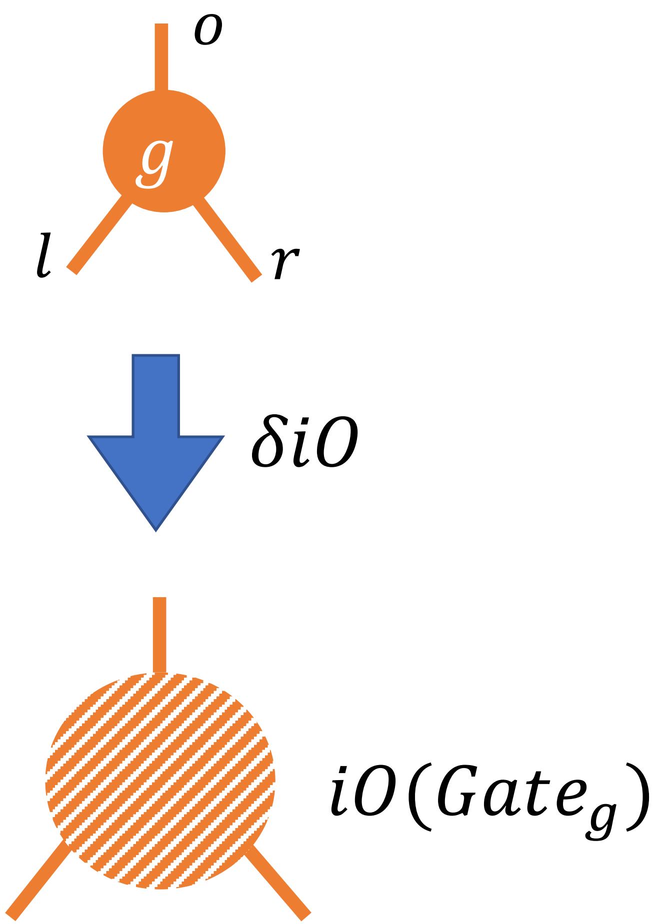
Technical Details

- $\mathcal{EF}$-Proofs $\Rightarrow$ δ-Equivalence
- **Construct $\boldsymbol{\delta iO}$**
- iO for Turing machines

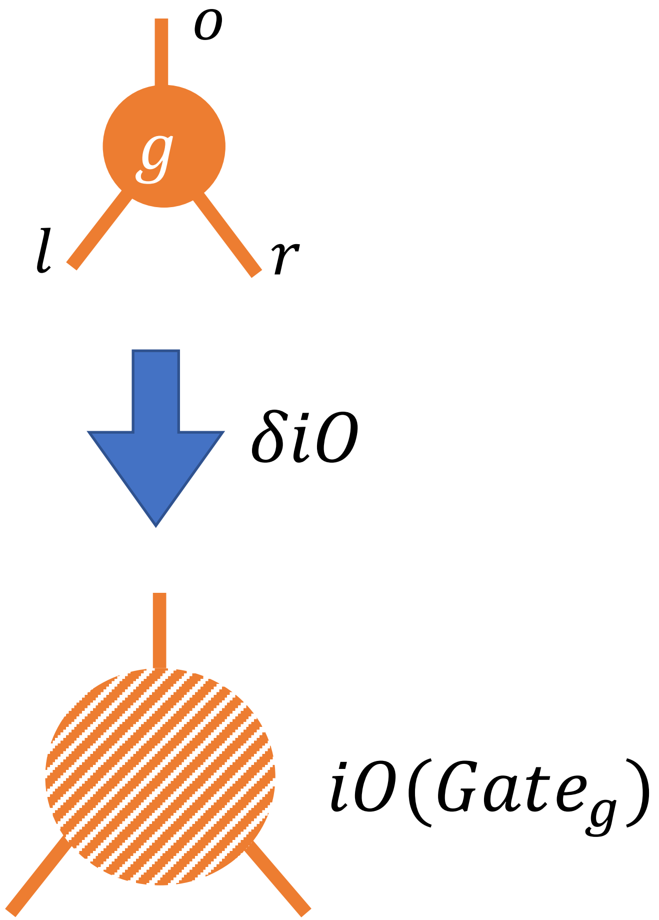
Gate-by-Gate Obfuscation

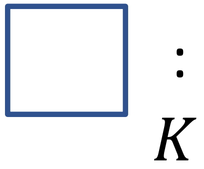


Gate-by-Gate Obfuscation



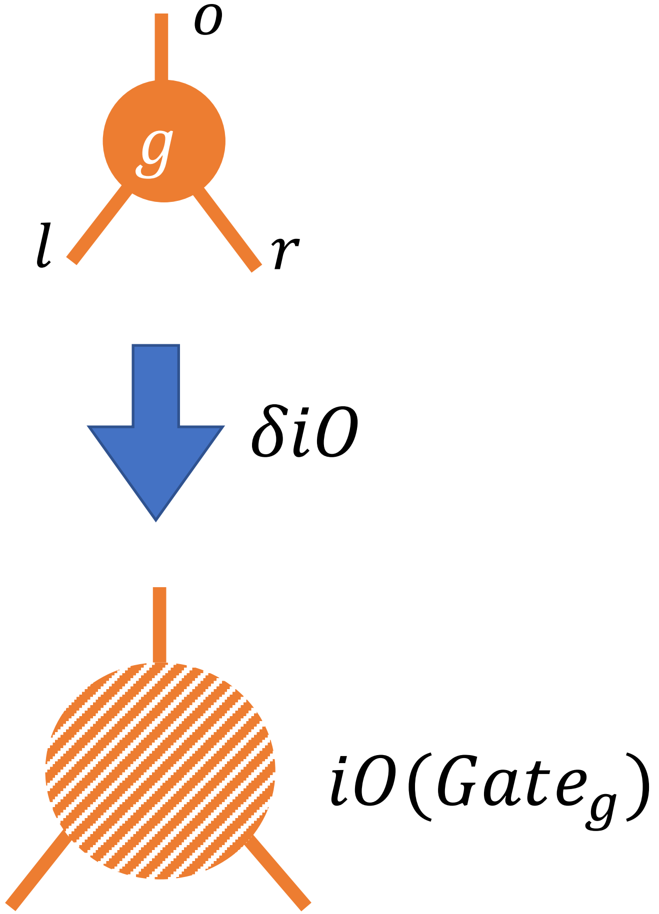
Gate-by-Gate Obfuscation





: Secret key encryption under key K

Gate-by-Gate Obfuscation



$\boxed{}_K$: Secret key encryption under key K

$$Gate_g(\ \boxed{m_l}_{K_l}\ \ \boxed{m_r}_{K_r}\ )$$

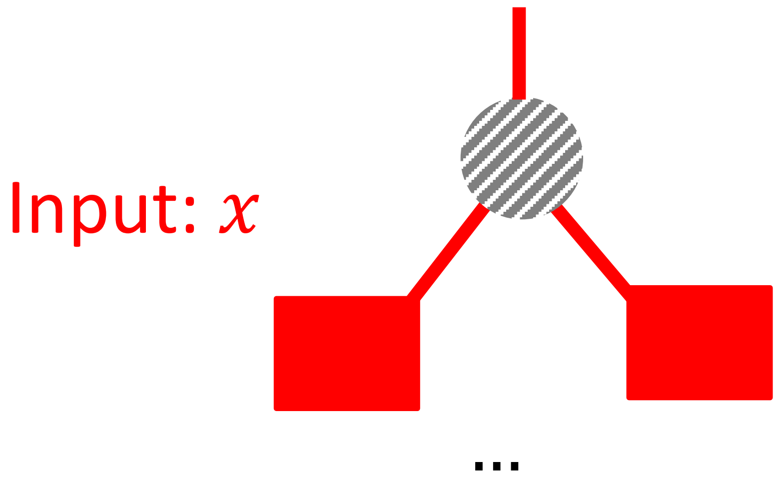
Decrypt m_l, m_r

$m_o = g(w_l, w_r)$

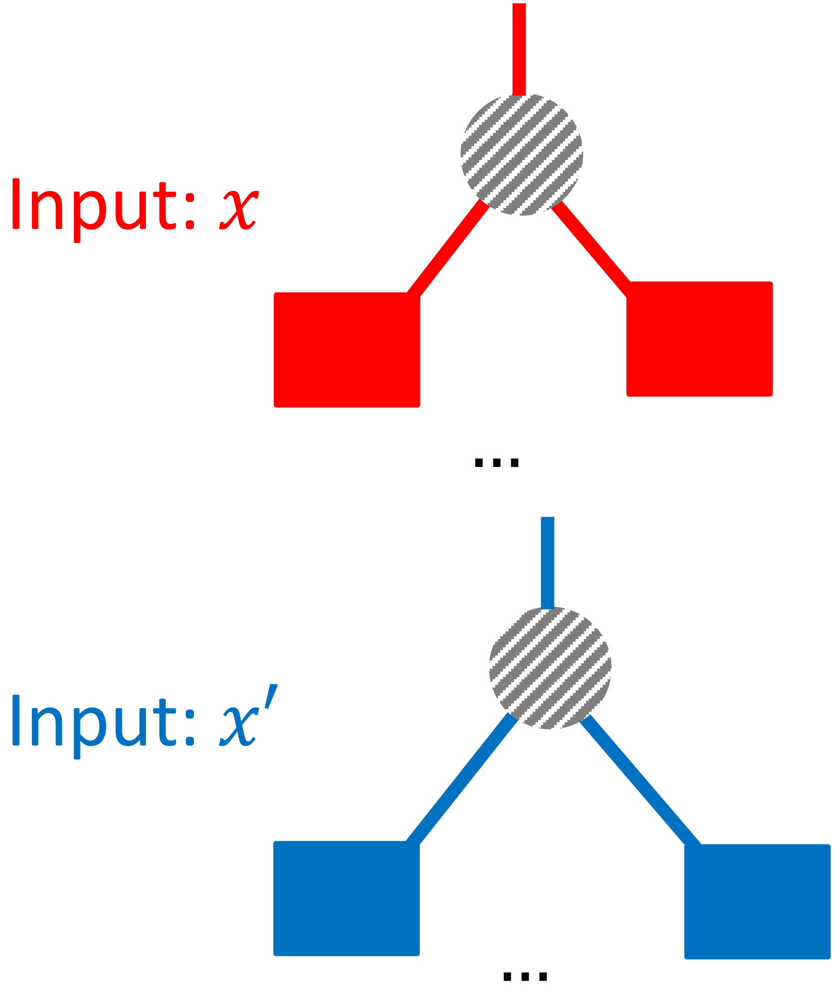
<u>Output</u>: $\boxed{m_o}_{K_o}$

Mix-and-Match Attack

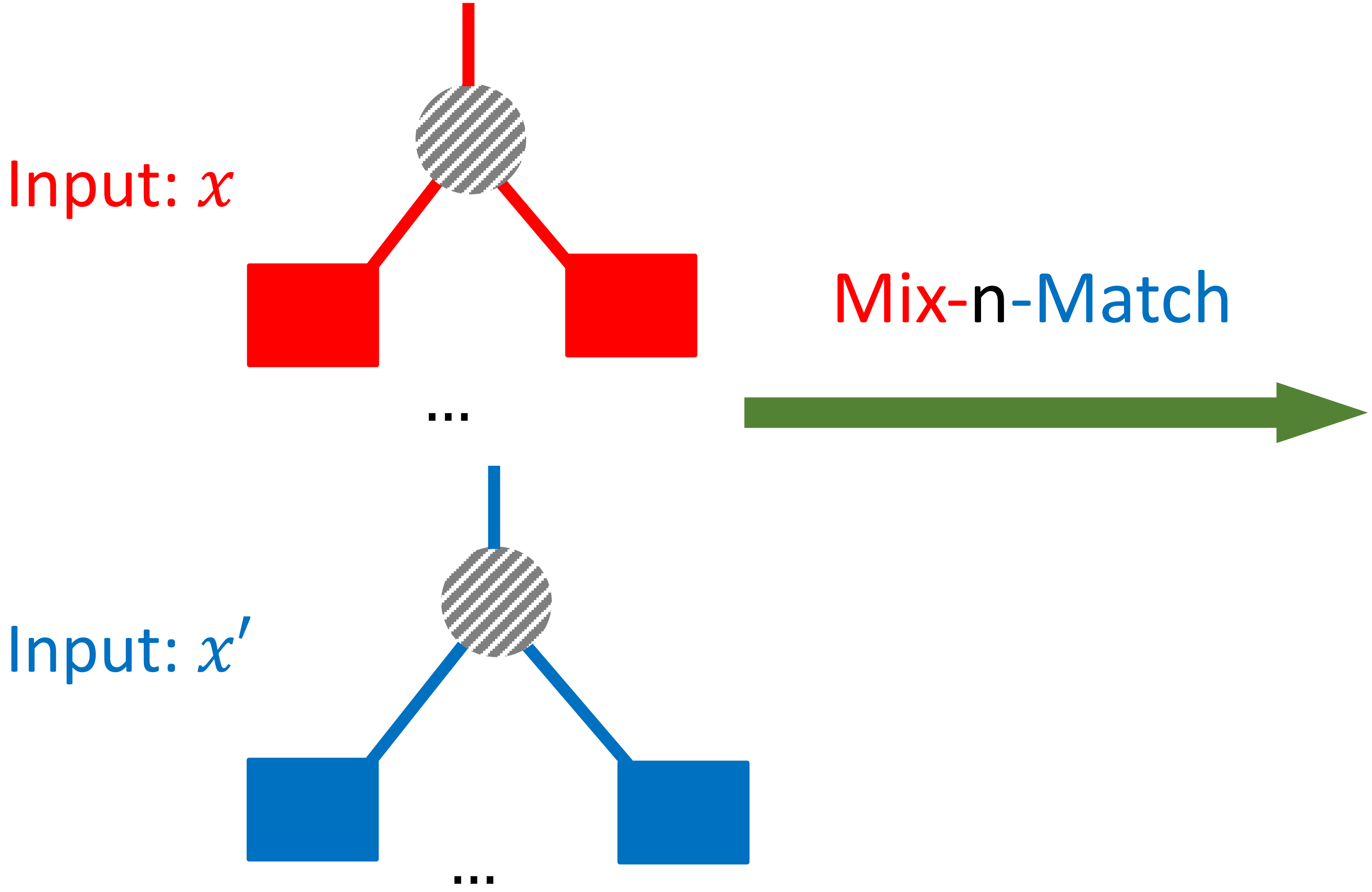
Mix-and-Match Attack



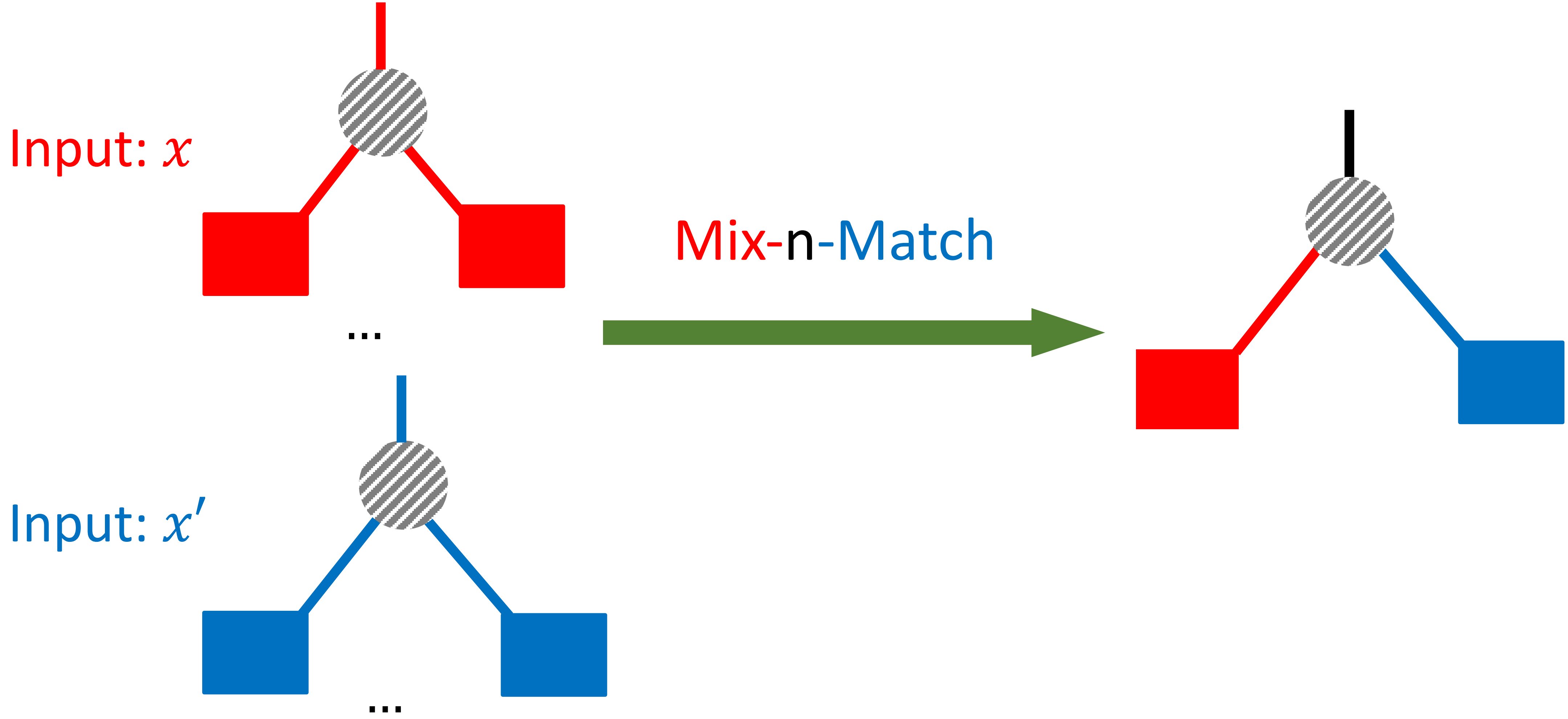
Mix-and-Match Attack



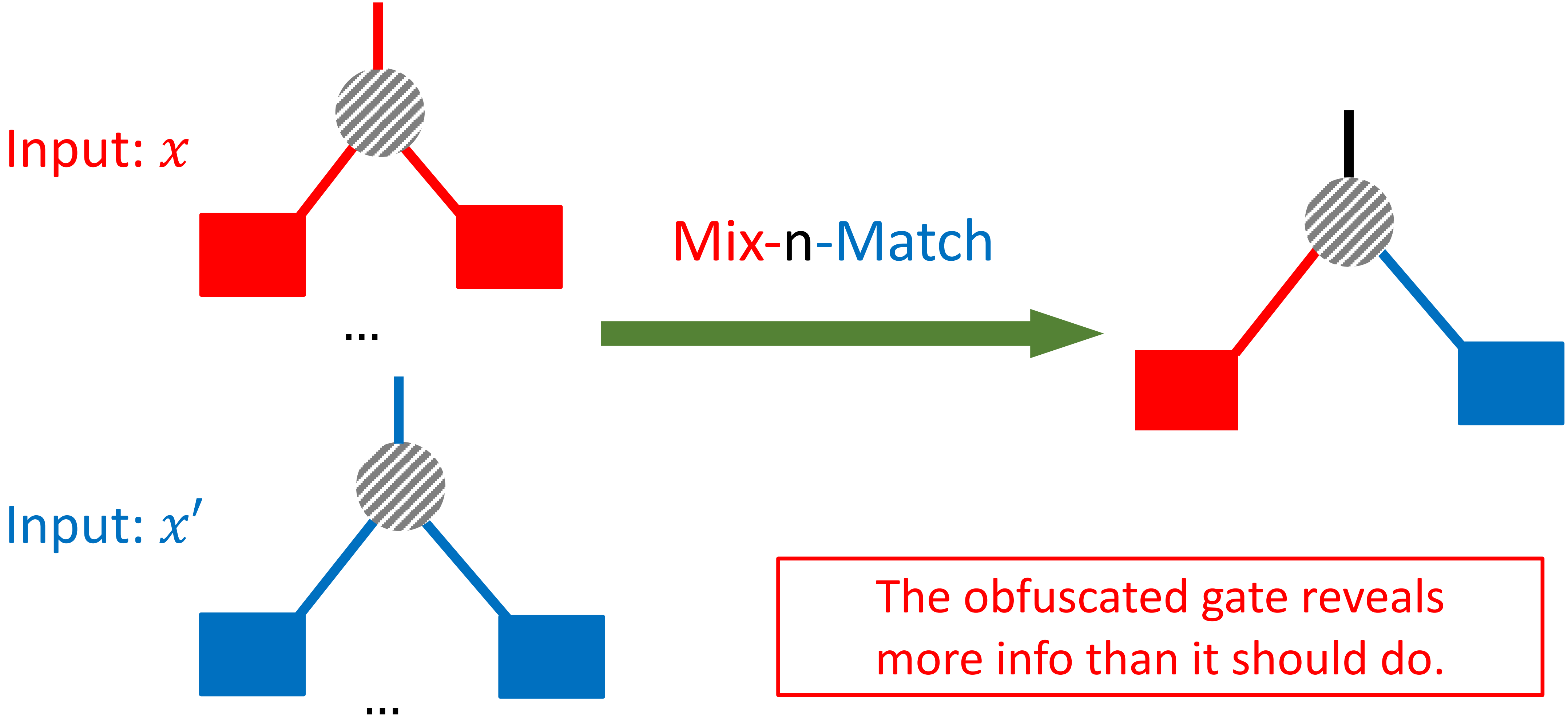
Mix-and-Match Attack



Mix-and-Match Attack



Mix-and-Match Attack



The obfuscated gate reveals more info than it should do.

Add Authentication

Add Authentication

$\forall$ wire w, sign ct_w with x:

$$\sigma_w := MAC_{K_w}(ct_w || x)$$

Add Authentication

$\forall$ wire w, sign ct_w with x:

$$\sigma_w \coloneqq MAC_{K_w}(ct_w || x)$$

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, x)}$$

Verify MAC σ_l, σ_r w.r.t. l, r

...(Decrypt, compute g, and re-encrypt)...

Also sign and output σ_o w.r.t. o

Add Authentication

$\forall$ wire w, sign ct_w with x:

$$\sigma_w := MAC_{K_w}(ct_w || x)$$

$$\frac{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, x)}{}$$

Verify MAC σ_l, σ_r w.r.t. l, r

...(Decrypt, compute g, and re-encrypt)...

Also sign and output σ_o w.r.t. o

x is too long!

Add Authentication

$\forall$ wire w, sign ct_w with x:

$$\sigma_w \coloneqq MAC_{K_w}(ct_w||x)$$

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, x)}$$

Verify MAC σ_l, σ_r w.r.t. l, r

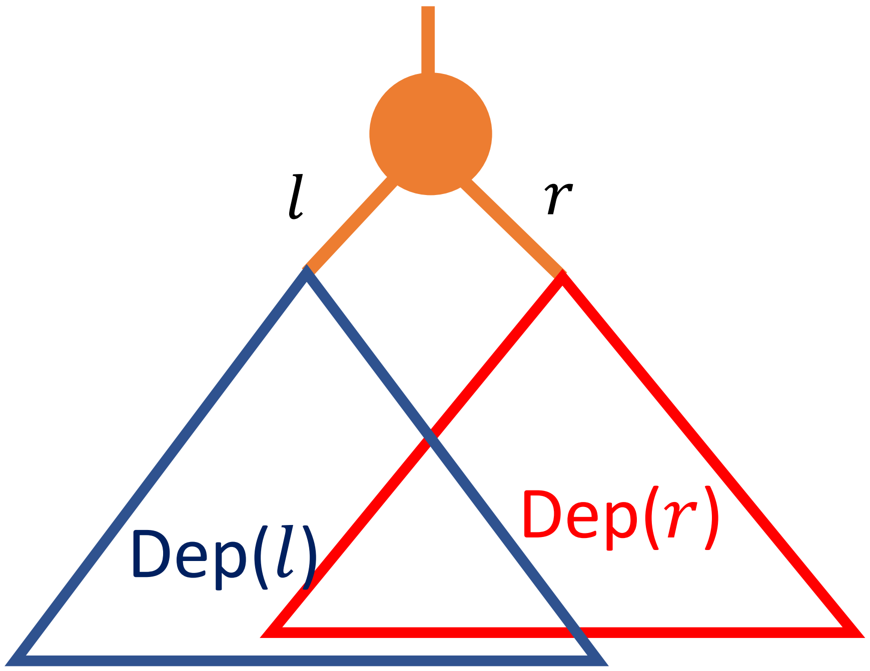
...(Decrypt, compute g, and re-encrypt)...

Also sign and output σ_o w.r.t. o

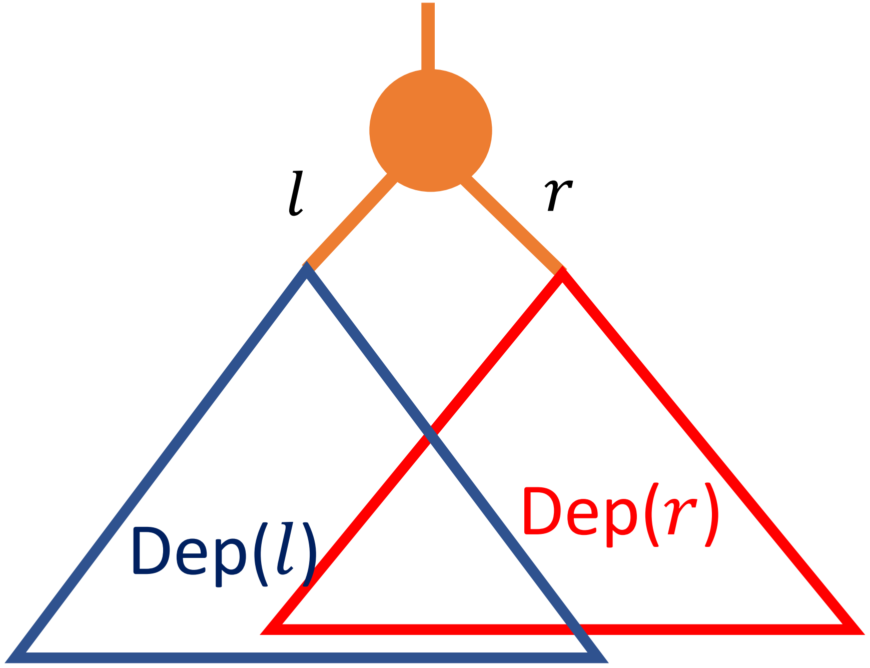
x is too long!

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

Define Dependence

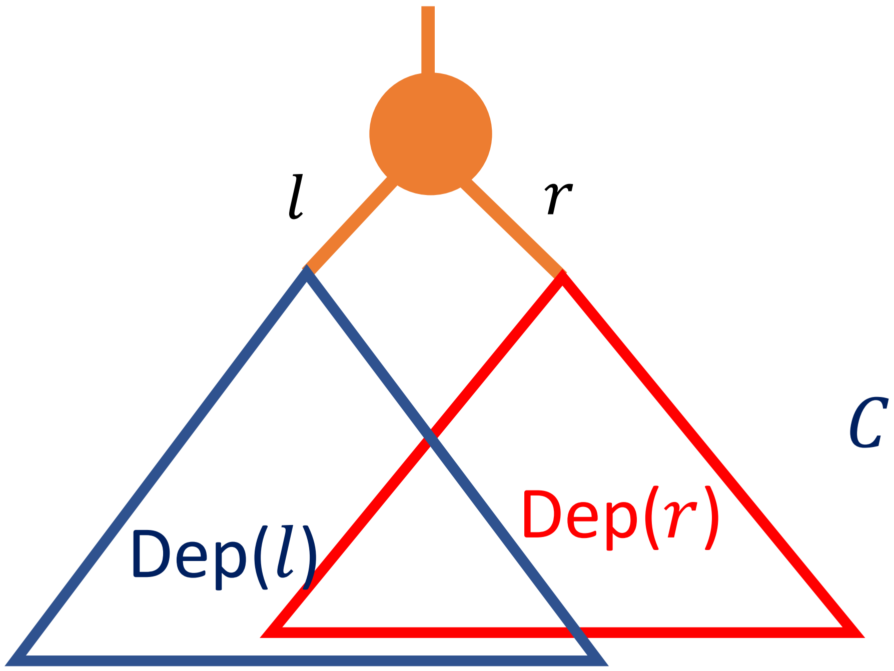


Define Dependence



$\mathrm{Dep}(l) \coloneqq \{w \mid l \text{ depends on wire } w\}$

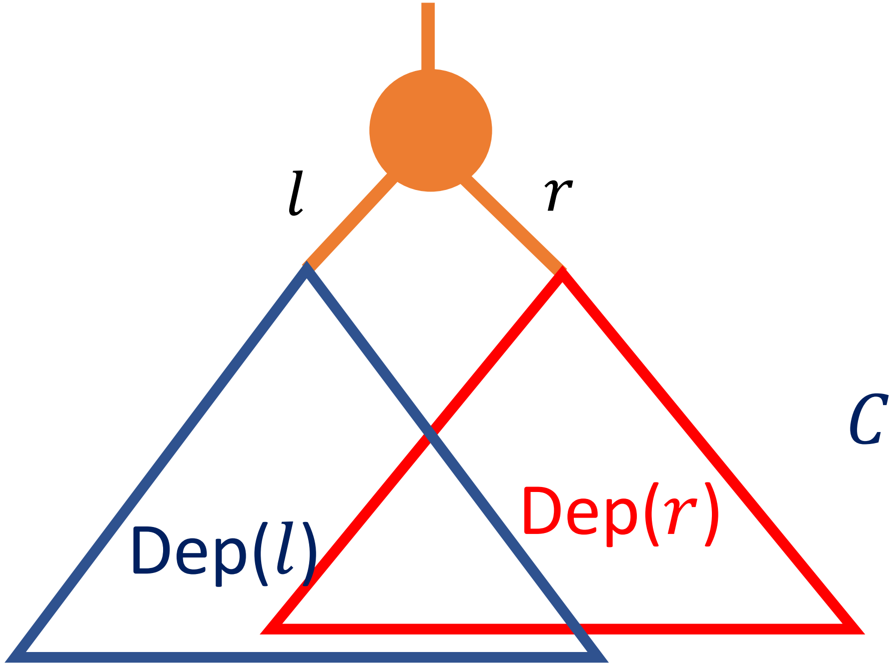
Define Dependence



$\mathrm{Dep}(l) := \{w \mid l \text{ depends on wire } w\}$

$CT_l := \{ciphertext\ of\ w\}_{w \in \mathrm{Dep}(l)}$ (An Index Set)

Define Dependence

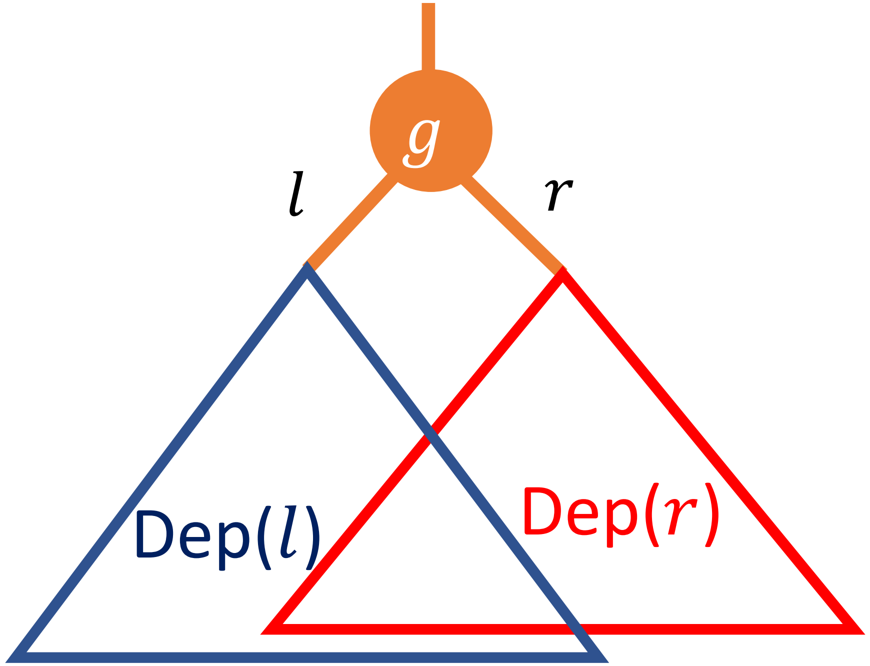


$\mathrm{Dep}(l) \coloneqq \{w \mid l \text{ depends on wire } w\}$

$CT_l \coloneqq \{ciphertext\ of\ w\}_{w \in \mathrm{Dep}(l)}$ (An Index Set)

($\mathrm{Dep}(r), CT_r$: Similar)

Use CT_l, CT_r in $Gate_g$



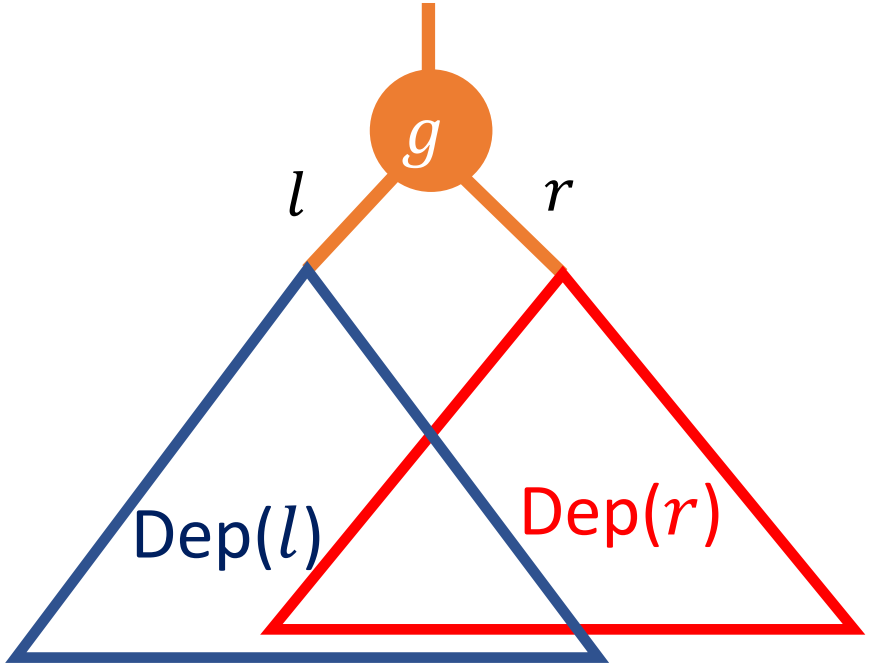
$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)}$$

Check $\sigma_l =^? MAC_{k_l}(ct_l || CT_l)$

Check $\sigma_r =^? MAC_{k_r}(ct_r || CT_r)$

...(Decrypt, compute g, and re-encrypt)...

Use CT_l, CT_r in $Gate_g$



$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)}$$

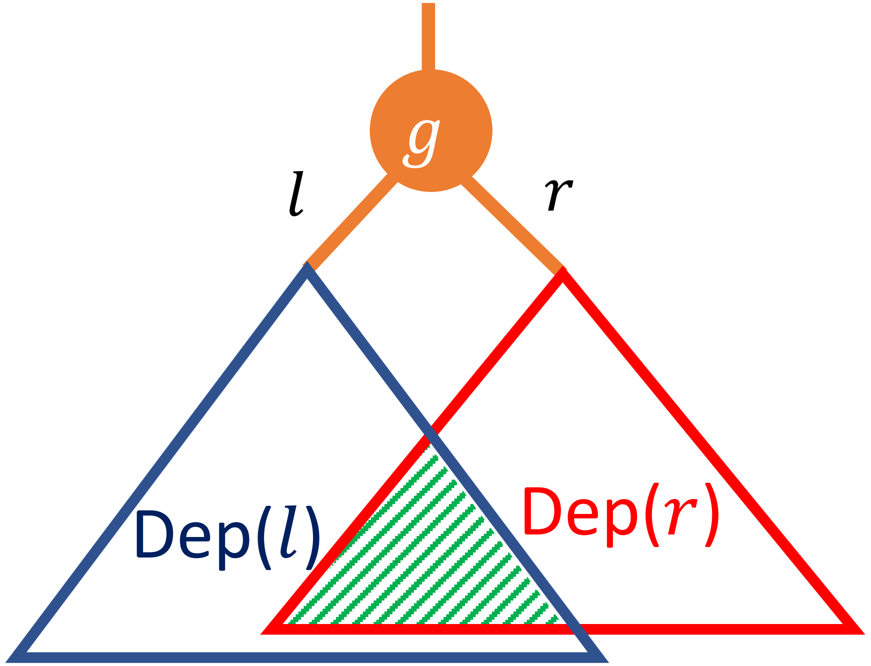
Check $\sigma_l =^? MAC_{k_l}(ct_l || CT_l)$

Check $\sigma_r =^? MAC_{k_r}(ct_r || CT_r)$

Check ***consistency*** of CT_l and CT_r

...(Decrypt, compute g, and re-encrypt)...

Use CT_l, CT_r in $Gate_g$



$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)}$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l || CT_l)$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r || CT_r)$

Check ***consistency*** of CT_l and CT_r

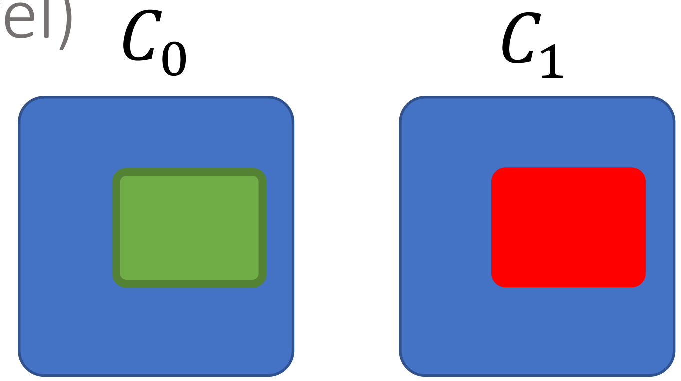
...(Decrypt, compute g, and re-encrypt)...

<u>CT_l and CT_r **are *Consistent***:</u>

CT_l, CT_r contains same ciphertexts in $\mathrm{Dep}(l) \cap \mathrm{Dep}(r)$

Proof of Security (High Level)

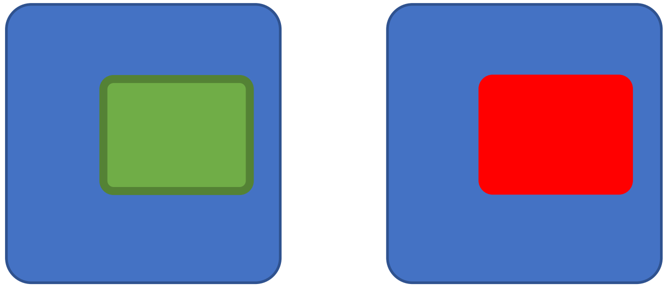
For any **$\boldsymbol{\delta}$-Equivalent** Ckts:



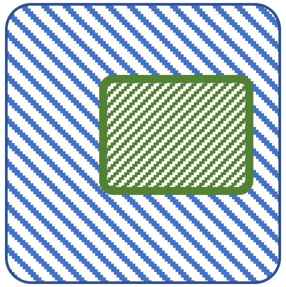
Proof of Security (High Level)

For any ***δ*-Equivalent** Ckts:

C_0 C_1



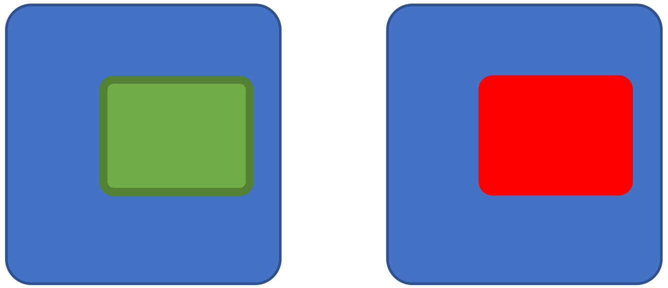
$\delta iO(C_0)$



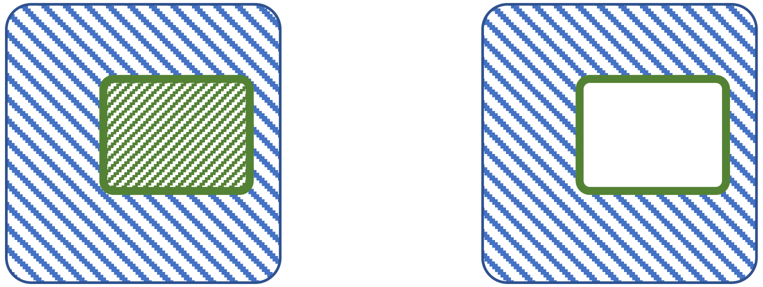
Proof of Security (High Level)

For any ***δ*-Equivalent** Ckts:

C_0 C_1



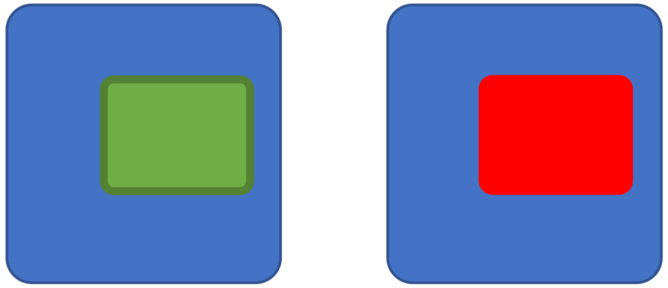
$\delta iO(C_0)$



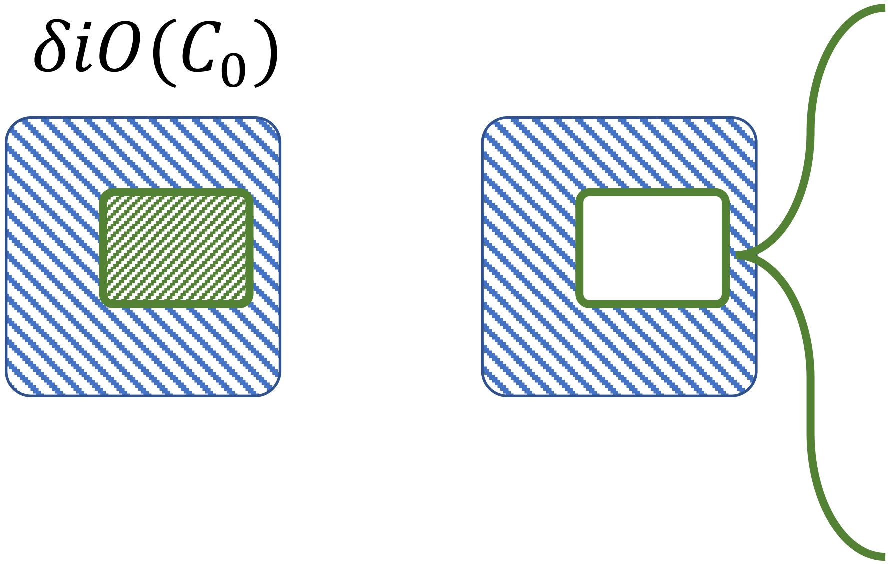
Proof of Security (High Level)

For any ***δ*-Equivalent** Ckts:

C_0 C_1



$\delta iO(C_0)$

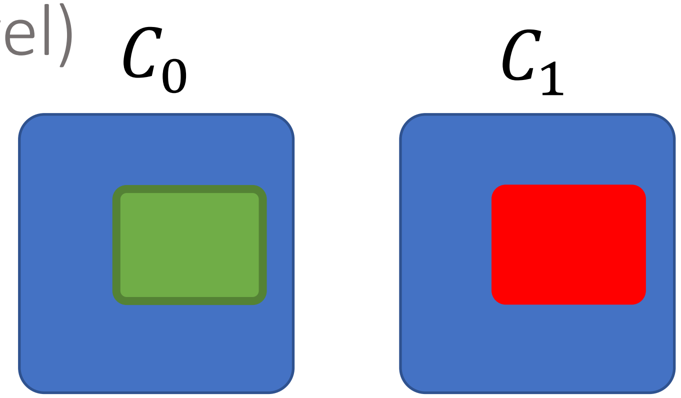


$\boldsymbol{Direct}\text{-}Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)$

...(check MACs & consistency)...

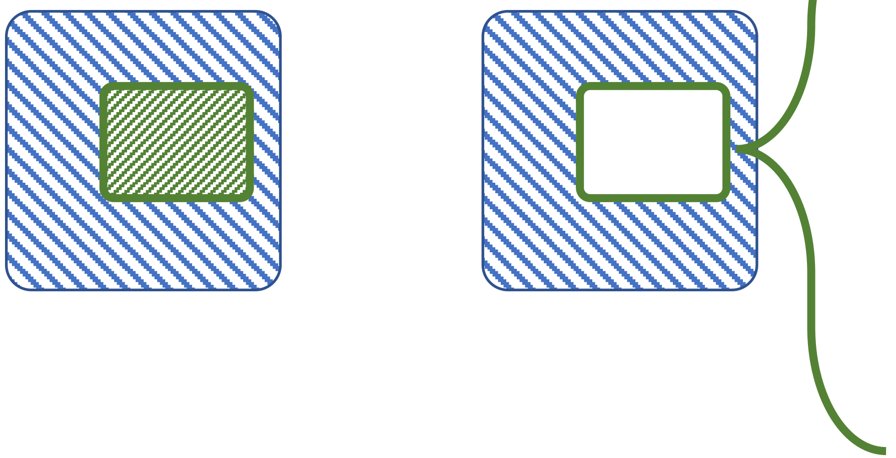
...(encrypt output wire)...

Proof of Security (High Level)



For any **$\boldsymbol{\delta}$-Equivalent** Ckts:

$\delta iO(C_0)$



$$\frac{\textbf{\textit{Direct}}\text{-}Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)}{}$$

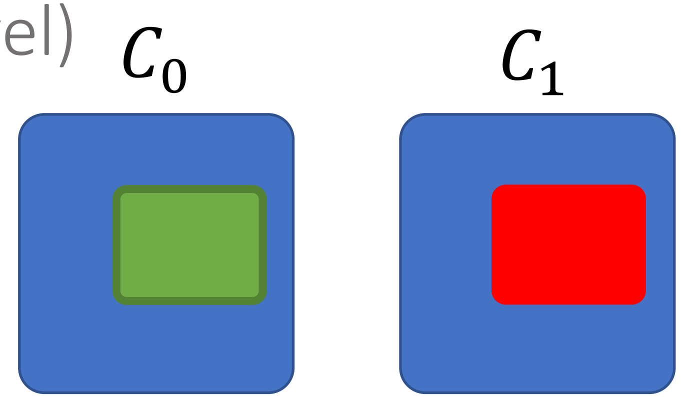
...(check MACs & consistency)...

Sub-ckt.input ← Decryt (CT_l, CT_r)

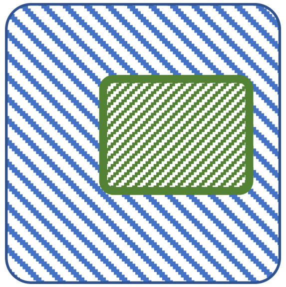
...(encrypt output wire)...

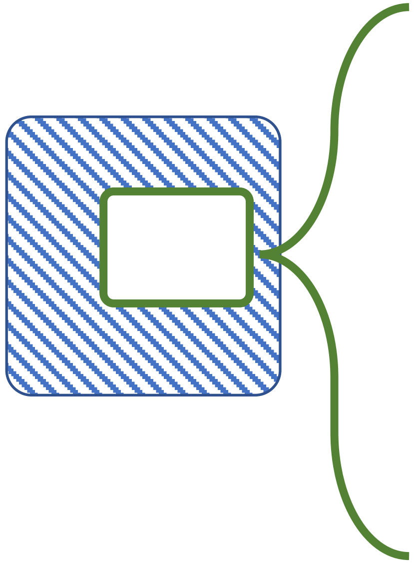
Proof of Security (High Level)

For any **δ-Equivalent** Ckts:



$\delta iO(C_0)$





$$\frac{\textbf{\textit{Direct}}\text{-}Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)}{}$$

...(check MACs & consistency)...

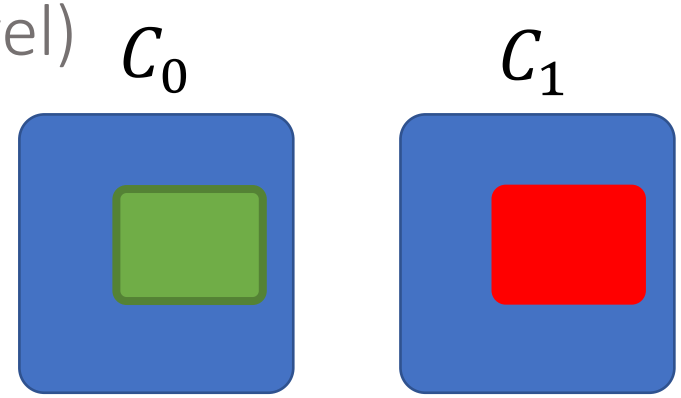
Sub-ckt.input ← Decryt (CT_l, CT_r)

Directly Compute Sub-ckt(sub-ckt.input)

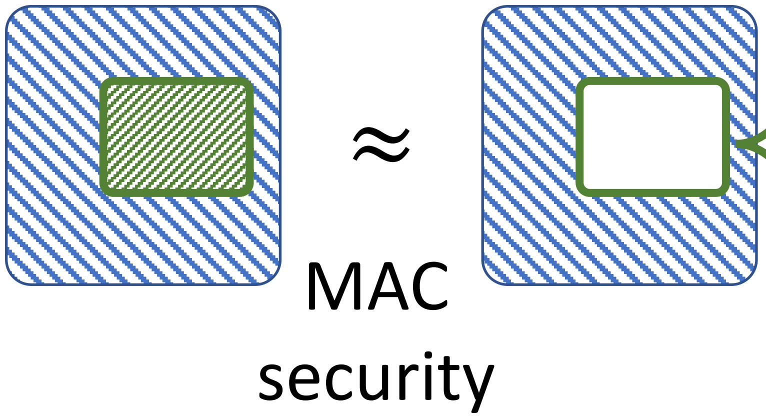
...(encrypt output wire)...

Proof of Security (High Level)

For any **$\boldsymbol{\delta}$-Equivalent** Ckts:



$\delta iO(C_0)$



$\boldsymbol{Direct}\text{-}Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)$

...(check MACs & consistency)...

Sub-ckt.input ← Decryt (CT_l, CT_r)

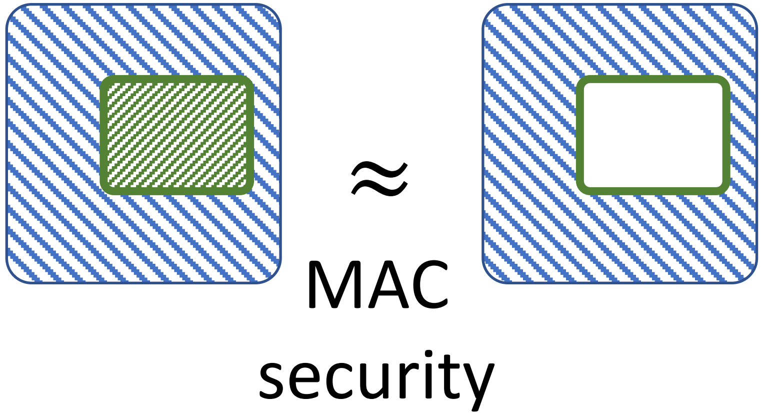
Directly Compute Sub-ckt(sub-ckt.input)

...(encrypt output wire)...

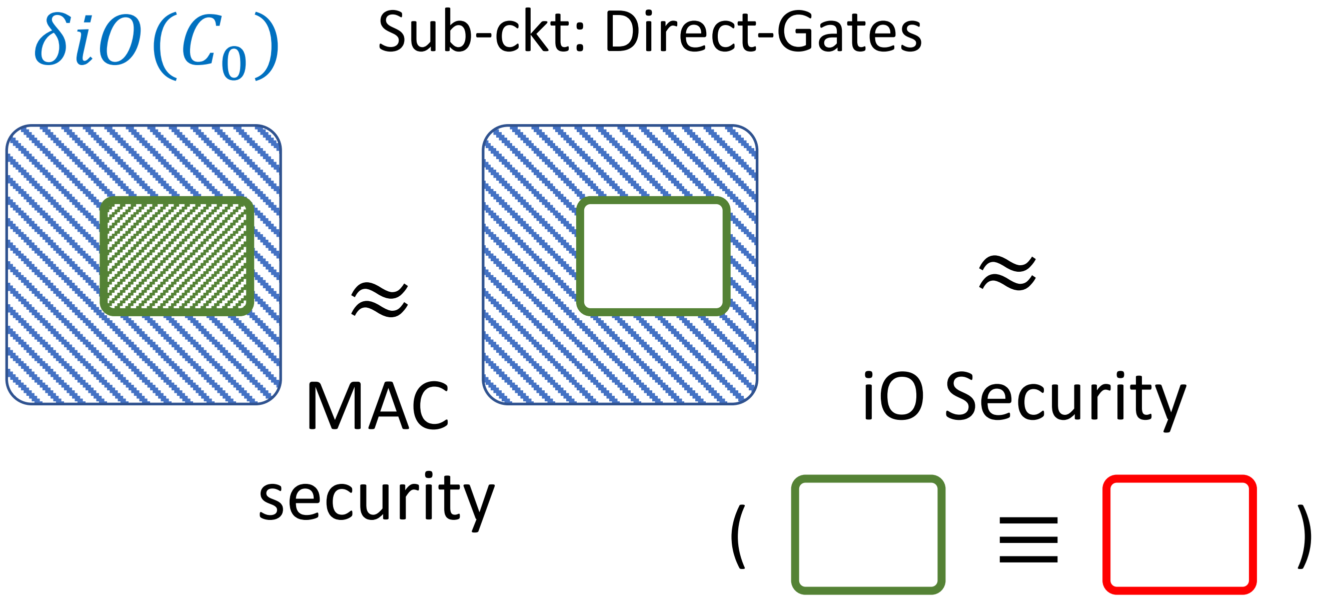
Proof of Security (High Level)

$\delta iO(C_0)$

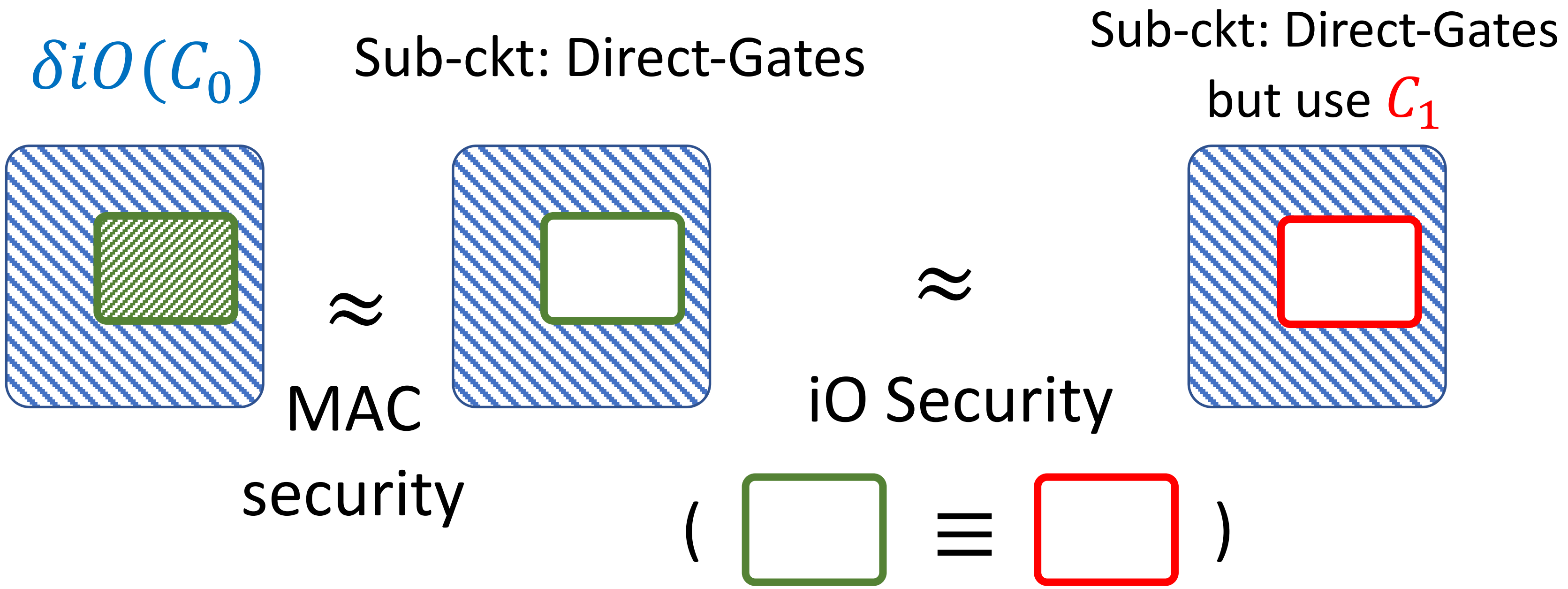
Sub-ckt: Direct-Gates



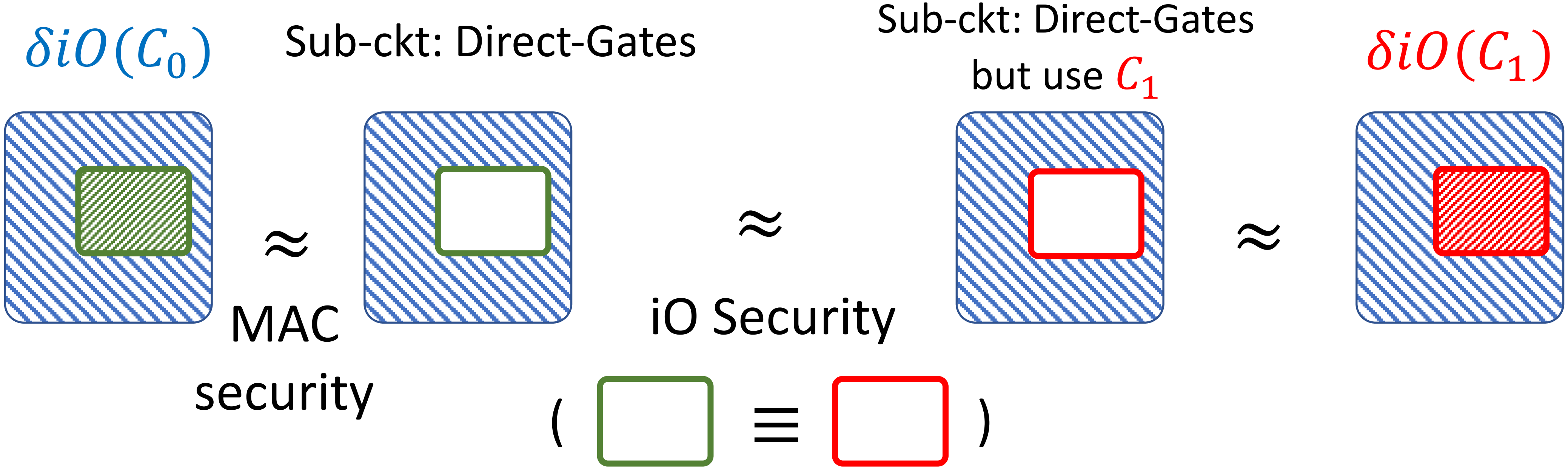
Proof of Security (High Level)



Proof of Security (High Level)

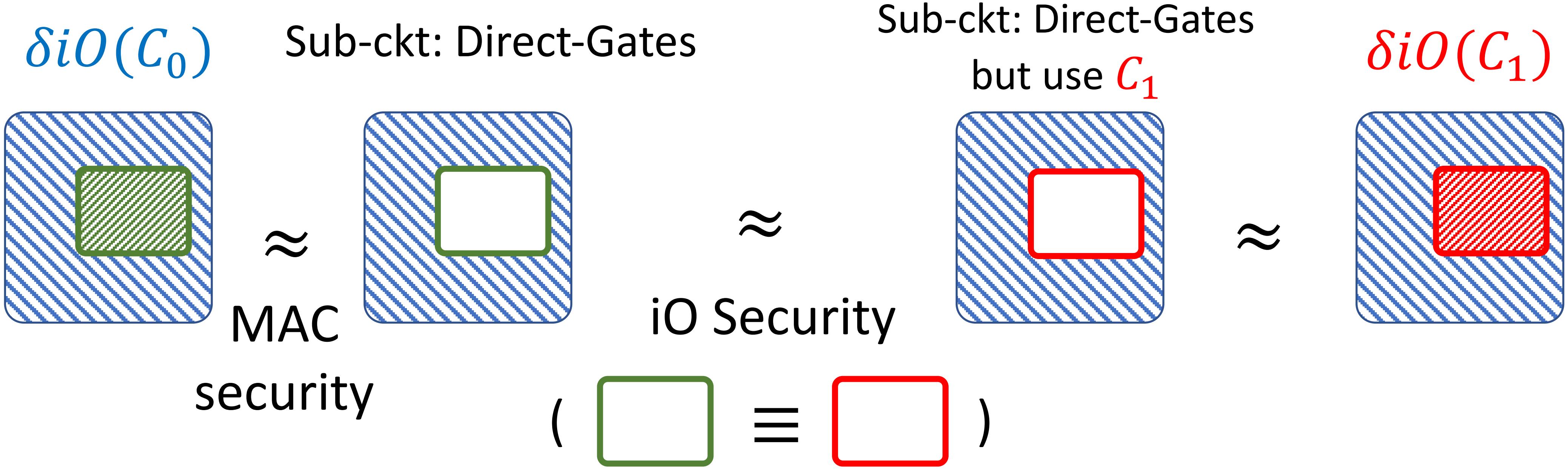


Proof of Security (High Level)



Proof of Security (High Level)

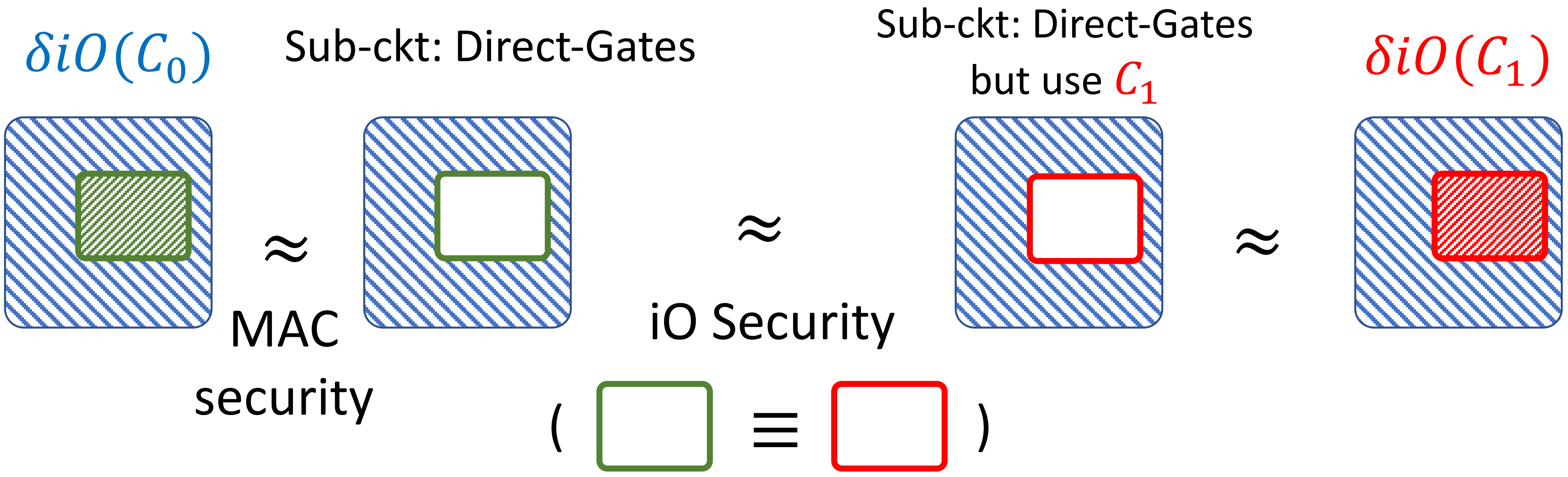
Extend this idea to general circuits?
Challenge: $|CT_l|$ is too large.



Proof of Security (High Level)

Extend this idea to general circuits?
Challenge: $|CT_l|$ is too large.

Observation:
g only depends on
sub-ckt input



Somewhere Statistical Binding (SSB) Hash

[Hubacek-Wichs'15, Okamoto-Pietrzak-Waters-Wichs'15]

Somewhere Statistical Binding (SSB) Hash

[Hubacek-Wichs'15, Okamoto-Pietrzak-Waters-Wichs'15]

Normal Mode

K

Somewhere Statistical Binding (SSB) Hash

[Hubacek-Wichs'15, Okamoto-Pietrzak-Waters-Wichs'15]

Normal Mode		**Trapdoor Mode**
K	$\approx_c$	$K^*(S \subseteq [n])$

Somewhere Statistical Binding (SSB) Hash

[Hubacek-Wichs'15, Okamoto-Pietrzak-Waters-Wichs'15]

Normal Mode		**Trapdoor Mode**
K	$\approx_c$	$K^*(S \subseteq [n])$

$$h \leftarrow \mathrm{SSB}(K, m_1, m_2, \ldots, m_n)$$

Somewhere Statistical Binding (SSB) Hash

[Hubacek-Wichs'15, Okamoto-Pietrzak-Waters-Wichs'15]

Normal Mode K $\approx_c$ **Trapdoor Mode** $K^*(S \subseteq [n])$

$h \leftarrow \mathrm{SSB}(K, m_1, m_2, \ldots, m_n)$

$h \leftarrow \mathrm{SSB}(K^*, m_1, m|_S, \ldots, m_n)$

Somewhere Statistical Binding (SSB) Hash

[Hubacek-Wichs'15, Okamoto-Pietrzak-Waters-Wichs'15]

Normal Mode

K

$\approx_c$

Trapdoor Mode

$K^*(S \subseteq [n])$

$h \leftarrow \mathrm{SSB}(K, m_1, m_2, \ldots, m_n)$

$h \leftarrow \mathrm{SSB}(K^*, m_1, m|_S, \ldots, m_n)$

In Our Setting: ($S :=$ {input wires to sub−ckt})

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, CT_l, CT_r)}$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l || CT_l)$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r || CT_r)$

...(Decrypt, compute g, and re-encrypt)...

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \qquad)$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l || CT_l)$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r || CT_r)$

...(Decrypt, compute g, and re-encrypt)...

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \quad h_l, h_r \quad)$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l || CT_l)$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r || CT_r)$

...(Decrypt, compute g, and re-encrypt)...

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \quad h_l, h_r \ )$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l|| \qquad)$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r|| \qquad)$

...(Decrypt, compute g, and re-encrypt)...

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \quad h_l, h_r \quad)}$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l || \; h_l \;)$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r || \; h_r \;)$

...(Decrypt, compute g, and re-encrypt)...

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \quad h_l, h_r \quad)}$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l || \ h_l \ )$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r || \ h_r \ )$

Check consistency of CT_l and CT_r

...(Decrypt, compute g, and re-encrypt)...

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \quad h_l, h_r \quad)}$$

Check $\sigma_l \stackrel{?}{=} MAC_{k_l}(ct_l || \ h_l \ )$

Check $\sigma_r \stackrel{?}{=} MAC_{k_r}(ct_r || \ h_r \ )$

Check consistency of CT_l and CT_r**???**

...(Decrypt, compute g, and re-encrypt)...

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \quad h_l, h_r \quad)}$$

Check $\sigma_l =^? MAC_{k_l}(ct_l || \; h_l \;)$

Check $\sigma_r =^? MAC_{k_r}(ct_r || \; h_r \;)$

Check consistency of CT_l and CT_r**???**

...(Decrypt, compute g, and re-encrypt)...

SNARGs?

No Statistical Soundness

SSB Hash CT_l, CT_r

Outside $Gate_g$:

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

$$Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, \quad h_l, h_r \ )$$

Check $\sigma_l =^? MAC_{k_l}(ct_l || \ h_l \ )$

Check $\sigma_r =^? MAC_{k_r}(ct_r || \ h_r \ )$

Check consistency of CT_l and CT_r???

...(Decrypt, compute g, and re-encrypt)...

SNARGs?
No Statistical Soundness

Consistency for sub-ckt input
(binding positions) is enough

Recall: SNARGs for Batch-Index

[Choudhuri-Jain-Jin'21]

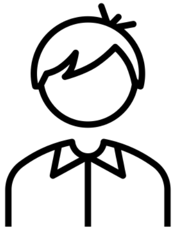
Index Language: $L = \{i | \exists w: C(i, w) = 1\}$

Recall: SNARGs for Batch-Index

[Choudhuri-Jain-Jin'21]

Index Language: $L = \{i | \exists w : C(i, w) = 1\}$

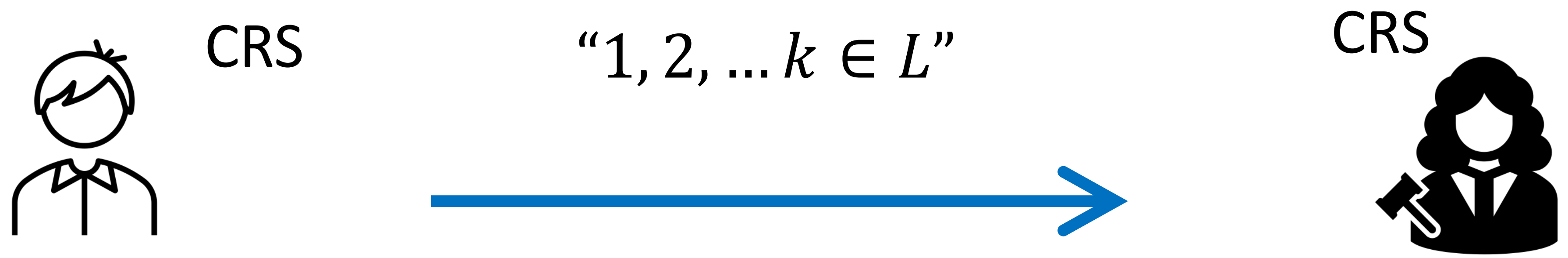
CRS



CRS

Recall: SNARGs for Batch-Index
[Choudhuri-Jain-Jin'21]

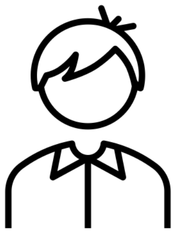
Index Language: $L = \{i | \exists w : C(i, w) = 1\}$



Recall: SNARGs for Batch-Index

[Choudhuri-Jain-Jin'21]

Index Language: $L = \{i | \exists w : C(i, w) = 1\}$



CRS

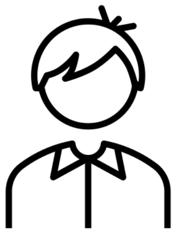
"$1, 2, \ldots k \in L$"

CRS

Verify in time $poly(\lambda, |C|, \log k)$

Recall: SNARGs for Batch-Index
[Choudhuri-Jain-Jin'21]

Index Language: $L = \{i | \exists w : C(i, w) = 1\}$



CRS

"$1, 2, \ldots k \in L$"

CRS

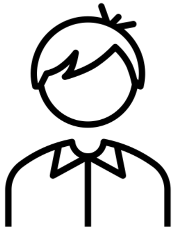
Verify in time $poly(\lambda, |C|, \log k)$

Accept/Reject

Recall: SNARGs for Batch-Index

[Choudhuri-Jain-Jin'21]

Index Language: $L = \{i | \exists w : C(i, w) = 1\}$



CRS

"$1, 2, \dots k \in L$"

CRS

Verify in time $poly(\lambda, |C|, \log k)$

Accept/Reject

Completeness:

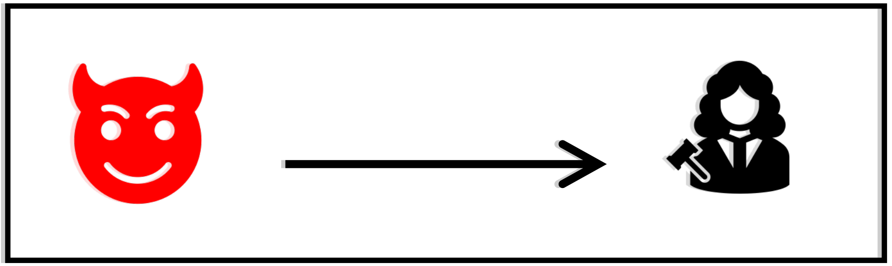
If $[k] \subseteq L$, honestly generated proof will be accepted.

Somewhere Statistical Soundness

Normal Mode		Trapdoor Mode
CRS	$\approx_c$	$CRS^*(S)$

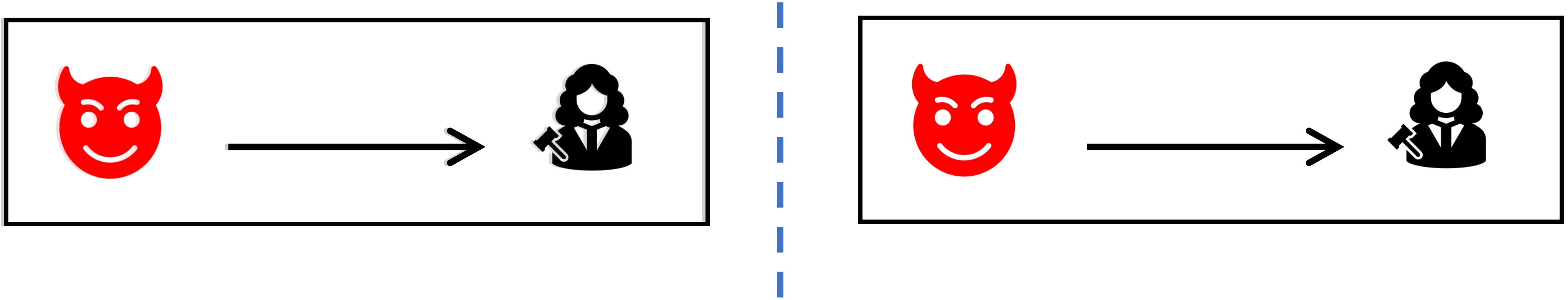
Somewhere Statistical Soundness

Normal Mode		**Trapdoor Mode**
CRS	$\approx_c$	$CRS^*(S)$



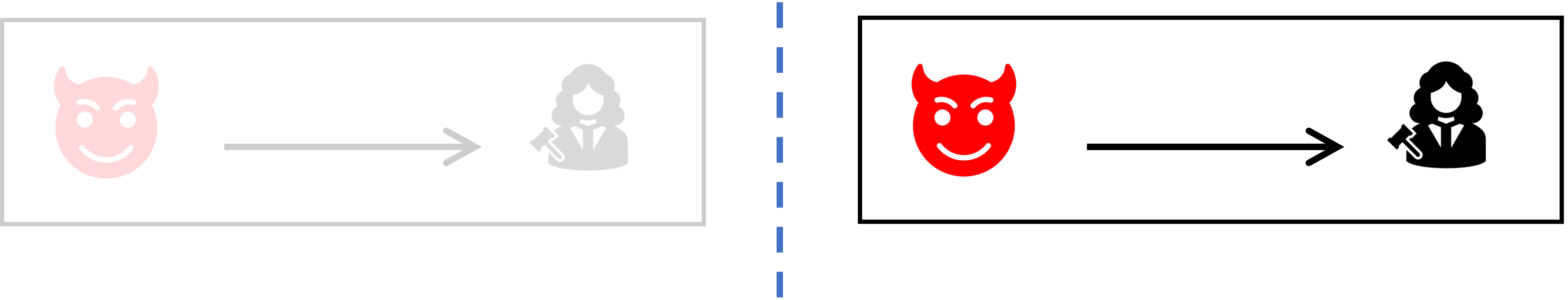
Somewhere Statistical Soundness

Normal Mode CRS $\approx_c$ **Trapdoor Mode** $CRS^*(S)$



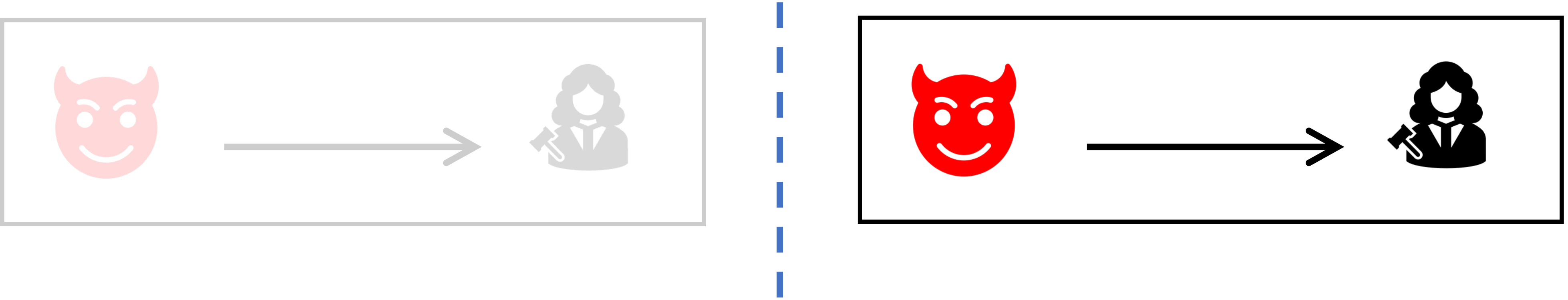
Somewhere Statistical Soundness

Normal Mode CRS $\approx_c$ **Trapdoor Mode** $CRS^*(S)$



Somewhere Statistical Soundness

Normal Mode		Trapdoor Mode
CRS	$\approx_c$	$CRS^*(S)$



Statistical Sound for $\boldsymbol{S}$:

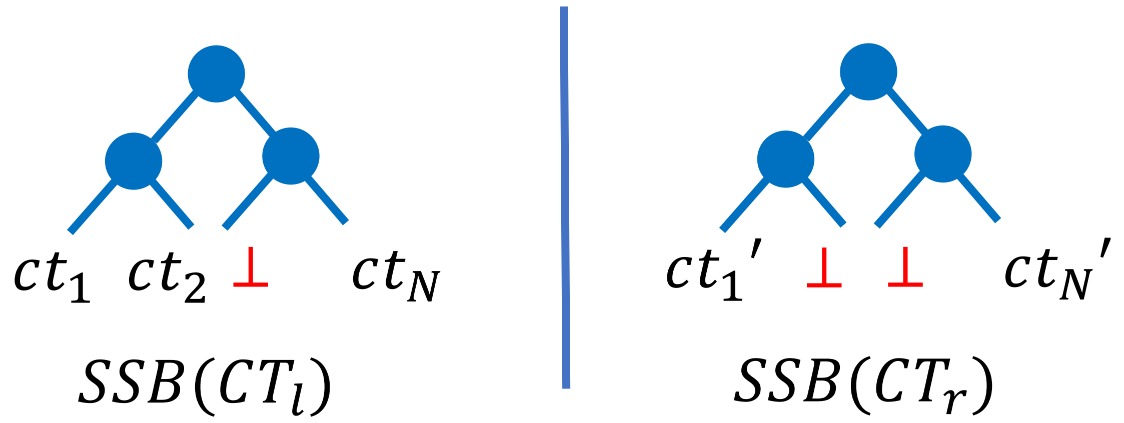
If $S \subseteq L$ does not hold, then

unbounded adv. can't find cheating proof.

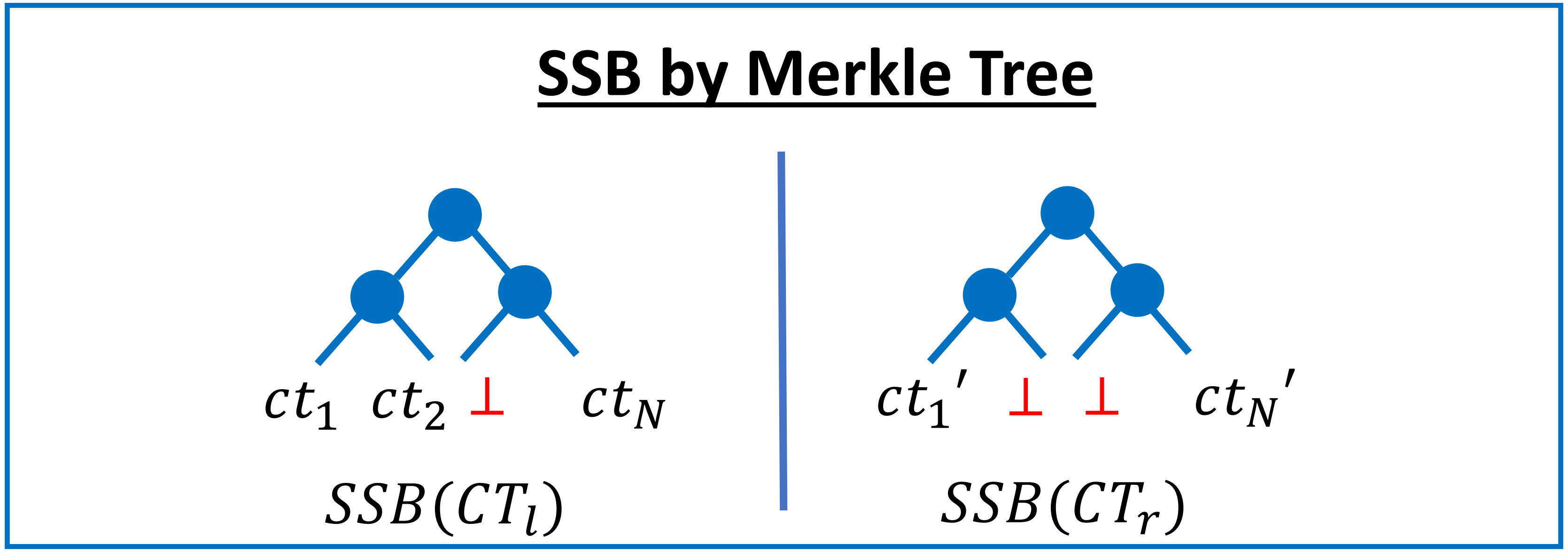
Construct Succinct Proof of Consistency

Construct Succinct Proof of Consistency

SSB by Merkle Tree



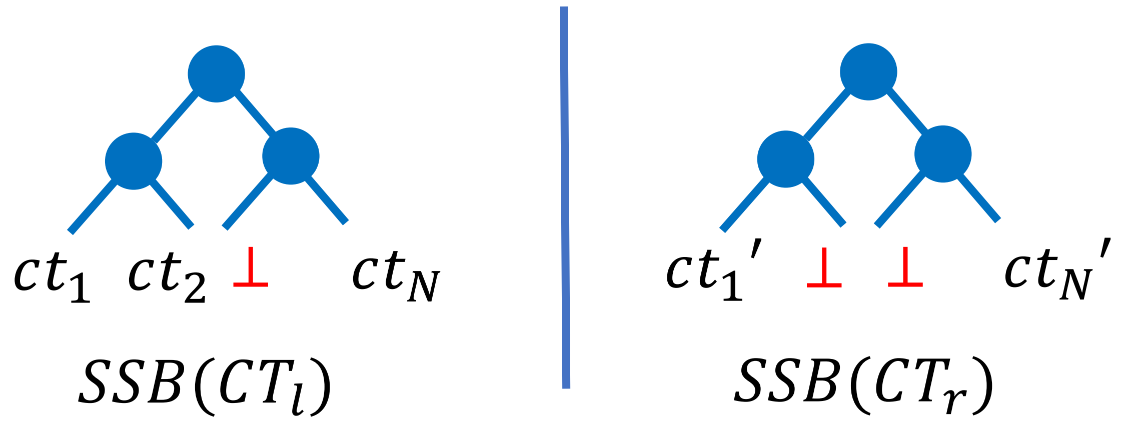
Construct Succinct Proof of Consistency



Prove: $\forall w \in [N]$, $\exists$ local openings & $ct_w, ct_w{}'$ s.t.
if $ct_w \neq \perp \wedge ct_w' \neq \perp$, then $ct_w = ct_w'$ ***(consistent)***

Construct Succinct Proof of Consistency

SSB by Merkle Tree



Prove: $\forall w \in [N]$, $\exists$ local openings & $ct_w, ct_w{}'$ s.t.
if $ct_w \neq \perp \wedge ct'_w \neq \perp$, then $ct_w = ct'_w$ ***(consistent)***

Proof via SNARGs for Batch-Index

Add Proof of Consistency

Outside $Gate_g$:

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

$$Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, h_l, h_r, \pi)$$

...(Verify the MACs)...

...(Decrypt, compute g, and re-encrypt)...

Add Proof of Consistency

Outside $Gate_g$:

$h_l = SSB(CT_l)$
$h_r = SSB(CT_r)$
π : consistency proof for h_l, h_r

$$\underline{Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, h_l, h_r, \pi)}$$

...(Verify the MACs)...

...(Decrypt, compute g, and re-encrypt)...

Add Proof of Consistency

Outside $Gate_g$:

$h_l = SSB(CT_l)$
$h_r = SSB(CT_r)$
π : consistency proof for h_l, h_r

$Gate_g(ct_l, ct_r, \sigma_l, \sigma_r, h_l, h_r, \pi)$

...(Verify the MACs)...

Verify the proof π w.r.t. h_l, h_r

...(Decrypt, compute g, and re-encrypt)...

Technical Details

- $\mathcal{EF}$-Proofs $\Rightarrow$ δ-Equivalence
- Construct δiO
- **iO for Turing machines**

Recall: Cook’s Translation

Recall: Cook's Translation

Poly-size $\mathcal{EF}$-Proofs $\xleftarrow{\text{Cook's Translation}}$ **Proofs in *PV***

Recall: Cook's Translation

Poly-size $\mathcal{EF}$-Proofs $\xleftarrow{\text{Cook's Translation}}$ **Proofs in *PV***

$$\vdash_{PV} M_1(x) = M_2(x)$$

Recall: Cook's Translation

Poly-size $\mathcal{EF}$-Proofs $\xleftarrow{\text{Cook's Translation}}$ **Proofs in *PV***

Input length n $\longleftarrow$ $\vdash_{PV} M_1(x) = M_2(x)$

Recall: Cook's Translation

Poly-size $\mathcal{EF}$-Proofs $\xleftarrow{\text{Cook's Translation}}$ **Proofs in *PV***

Input length n

$$\vdash_{\mathcal{EF}} C_{1,n}(x) \leftrightarrow C_{2,n}(x) \quad \longleftarrow \quad \vdash_{PV} M_1(x) = M_2(x)$$

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

Recall: Cook's Translation

Poly-size $\mathcal{EF}$-Proofs $\xleftarrow{\text{Cook's Translation}}$ **Proofs in *PV***

$$\vdash_{\mathcal{EF}} C_{1,n}(x) \leftrightarrow C_{2,n}(x) \xleftarrow{\text{Input length } n} \vdash_{PV} M_1(x) = M_2(x)$$

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

Recall: Cook's Translation

Poly-size $\mathcal{EF}$-Proofs	$\xleftarrow{\text{Cook's Translation}}$	Proofs in *PV*

Input length n

$$\vdash_{\mathcal{EF}} C_{1,n}(x) \leftrightarrow C_{2,n}(x) \quad \longleftarrow \quad \vdash_{PV} M_1(x) = M_2(x)$$

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

Use δiO?

iO for TMs from δiO

iO for TMs from δiO

M

Turing Machine

iO for TMs from δiO

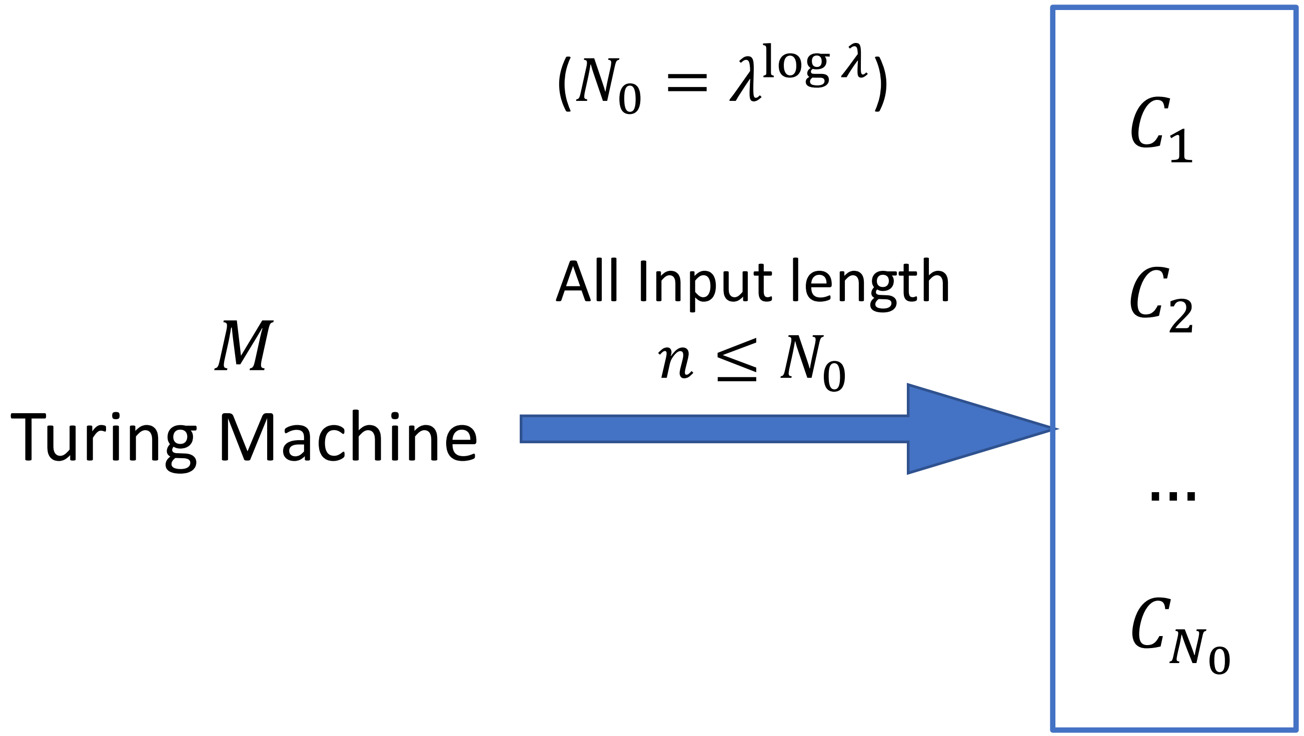
$(N_0 = \lambda^{\log \lambda})$

All Input length
$n \le N_0$

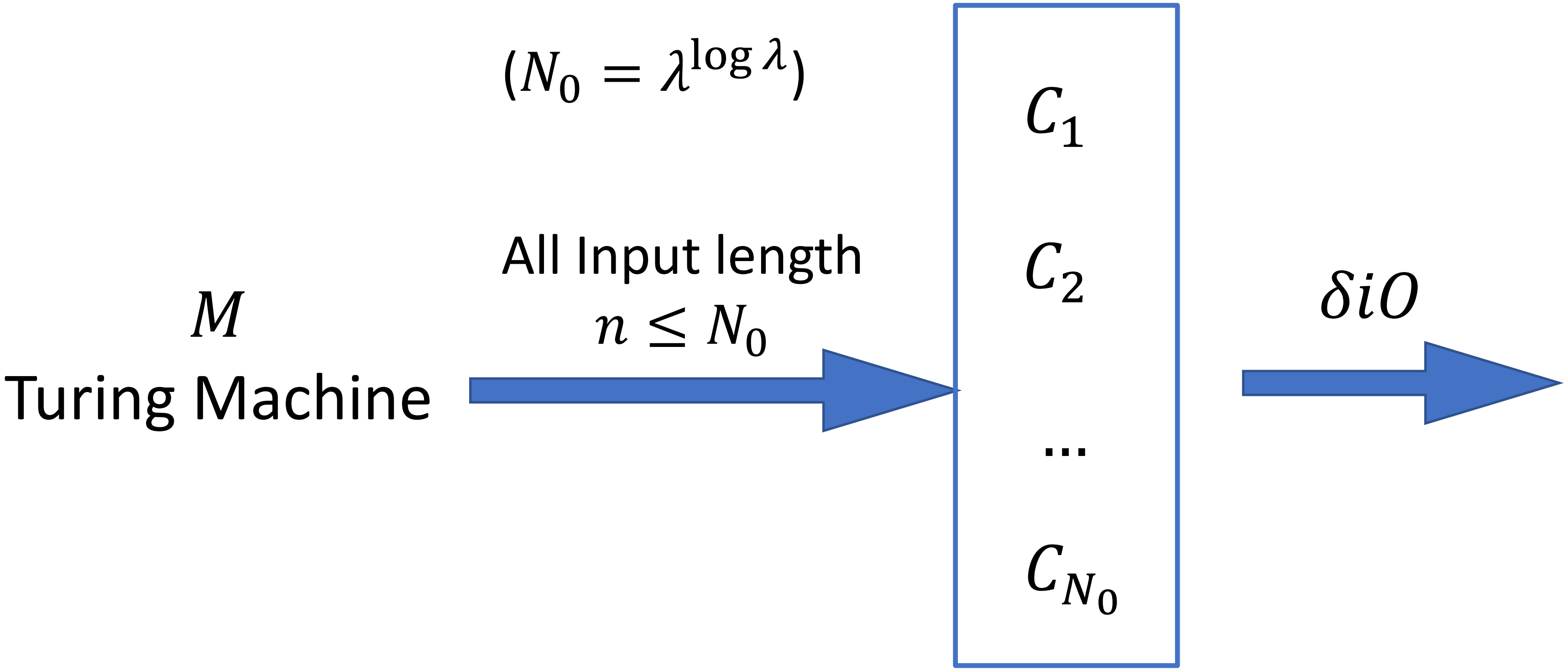
M
Turing Machine

iO for TMs from δiO

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



iO for TMs from δiO



iO for TMs from δiO

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

M

Turing Machine

All Input length

$n \leq N_0$

C_1

C_2

...

C_{N_0}

δiO

$\delta iO(C_1)$

$\delta iO(C_2)$

...

$\delta iO(C_{N_0})$

Obfuscated Program

iO for TMs from δiO

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

M
Turing Machine

All Input length
$n \le N_0$

C_1
C_2
...
C_{N_0}

δiO

$\delta iO(C_1)$
$\delta iO(C_2)$
...
$\delta iO(C_{N_0})$

Obfuscated Program

Obfuscation time is super-poly!

Efficient Construction

Efficient Construction

$C_1, C_2, \ldots, C_{N_0}$ have a succinct description

Efficient Construction

$C_1, C_2, \ldots, C_{N_0}$ have a succinct description

i.e. $\exists$ circuit $[M](\cdot,\cdot)$, s.t. $[M](n, i)$ outputs the description of i-th gate in C_n

Efficient Construction

$C_1, C_2, \ldots, C_{N_0}$ have a succinct description

i.e. $\exists$ circuit $[M](\cdot,\cdot)$, s.t. $[M](n, i)$ outputs the description of i-th gate in C_n

Recall: δiO Construction

$$g_1, g_2, \ldots, g_{|C|} \xrightarrow{\delta iO} \begin{matrix} iO(Gate_{g_1}) \\ iO(Gate_{g_2}) \\ \ldots \\ iO(Gate_{g_{|C|}}) \end{matrix}$$

Generate $Gate_g$ from $[M](\cdot,\cdot)$

Efficient Construction

Efficient Construction

M

Turing Machine

Efficient Construction

M
Turing Machine

Efficient Construction

M
Turing Machine

$\Rightarrow$

$UGate(\cdot,\cdot,\cdot)$

Efficient Construction

(iO for small circuit)

M
Turing Machine $\Rightarrow$ $UGate(\cdot,\cdot,\cdot)$ $\Rightarrow$ $iO(UGate)$

Efficient Construction

(iO for small circuit)

M
Turing Machine → $UGate(\cdot,\cdot,\cdot)$ → $iO(UGate)$

"Uniform" Gate $UGate(n, i, input')$

Get description of i-th gate:

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

Emulate

$$Gate_g(input')$$

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 $\Longleftrightarrow$ Techniques to argue Indistinguishability for iO

$\mathcal{EF}$ / PV $\Longleftrightarrow$ δ-equivalence & δiO

Summary & Future Directions

Inference Rules in **Logic systems** for Proving Equivalence ⟺ Techniques to argue Indistinguishability for iO

$\mathcal{EF}$ / PV ⟺ δ-equivalence & δiO

ZFC (<u>**Z**</u>ermelo-<u>**F**</u>raenkel set theory with axiom of <u>**C**</u>hoice) ⟺ **?**