The Usefulness of Sparsifiable Inputs: How to Avoid Subexponential iO

Thomas Agrikola[1] Geoffroy Couteau[2] Dennis Hofheinz[3]

[1]Karlsruhe Institute of Technology (KIT), Germany
[2]IRIF, Paris-Diderot University, CNRS, France
[3]ETH Zurich, Switzerland

May 12, 2020

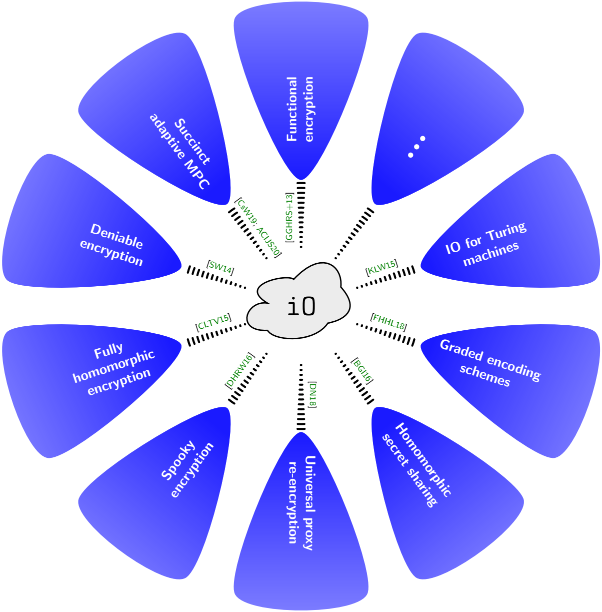
Introduction

Indistinguishability obfuscation (IO) is a method to transform a program into an *unintelligible* one maintaining the original functionality.

$$
\begin{tikzcd}
\boxed{P_1} \arrow[d, "\mathtt{iO}"'] & \equiv & \boxed{P_2} \arrow[d, "\mathtt{iO}"] \\
\boxed{\mathtt{iO}(P_1)} & \approx & \boxed{\mathtt{iO}(P_2)}
\end{tikzcd}
$$

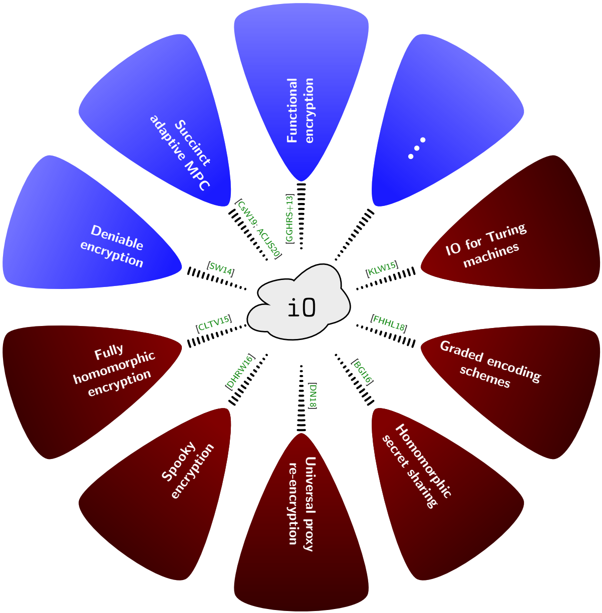
Applications of iO

- We can build almost anything from iO.



Applications of iO

- We can build almost anything from iO.
- But what can we do from **polynomial** iO?



poly reduction to iO	
subexp reduction to iO	

Related work on removing subexp. IO

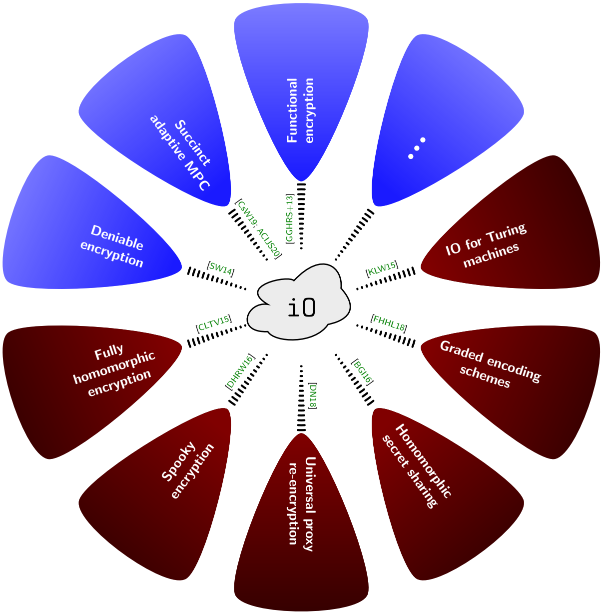
- Previous approaches to avoid subexponential reductions to iO: replace iO with functional encryption, [GS16; GPSZ17; LZ17; KLMR18]
  - short signatures
  - universal samplers
  - non-interactive multiparty key exchange
  - trapdoor one-way permutations
  - multi-key functional encryption
  - ...

Related work on removing subexp. IO

- Previous approaches to avoid subexponential reductions to iO: replace iO with functional encryption, [GS16; GPSZ17; LZ17; KLMR18]
  - short signatures
  - universal samplers
  - non-interactive multiparty key exchange
  - trapdoor one-way permutations
  - multi-key functional encryption
  - ...
- **But** the supported operations are relatively restricted

Applications of iO

- We can build almost anything from iO.
- But what can we do from **polynomial** iO?

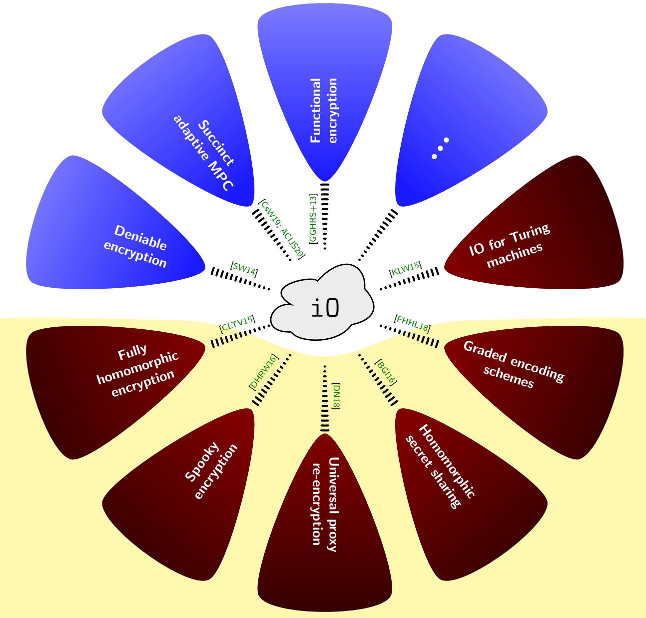


poly reduction to iO	
subexp reduction to iO	

Applications of iO

- We can build almost anything from iO.
- But what can we do from **polynomial** iO?

poly reduction to iO	
subexp reduction to iO	
piO abstraction	



Probabilistic IO

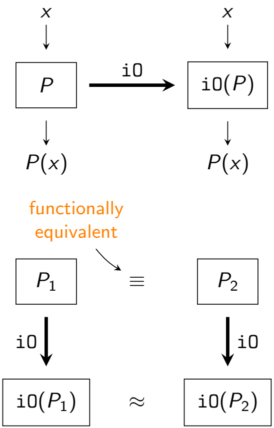
$\mathtt{iO}$ compiles programs into
unintelligible ones, while
preserving their functionality.

$$\begin{tikzcd}
x \arrow[d] & & x \arrow[d] \\
\boxed{P} \arrow[d] \arrow[rr, "\mathtt{iO}"] & & \boxed{\mathtt{iO}(P)} \arrow[d] \\
P(x) & & P(x)
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{P_1} \arrow[d, "\mathtt{iO}"'] & \equiv & \boxed{P_2} \arrow[d, "\mathtt{iO}"] \\
\boxed{\mathtt{iO}(P_1)} & \approx & \boxed{\mathtt{iO}(P_2)}
\end{tikzcd}$$

Probabilistic IO

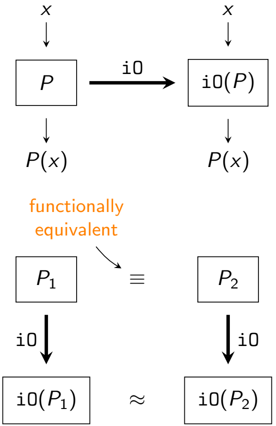
`iO` compiles programs into *unintelligible* ones, while
preserving their functionality.

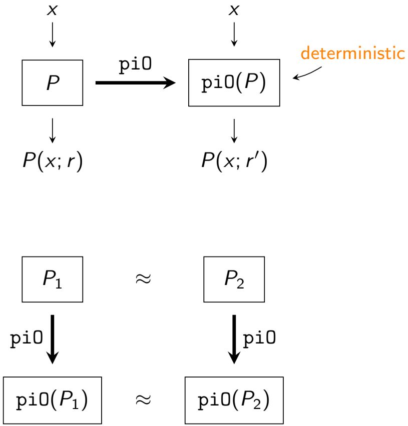


Probabilistic IO

`iO` compiles programs into *unintelligible* ones, while *preserving their functionality*.

`piO` compiles *randomized* programs into *deterministic unintelligible* ones, while *preserving their functionality*.

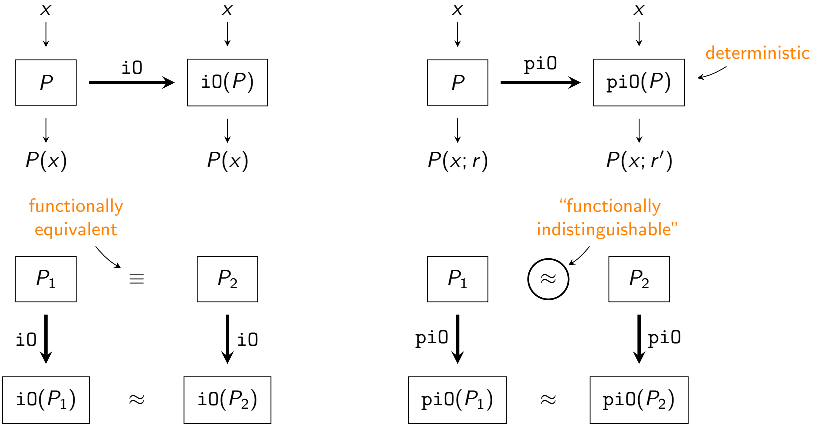




Probabilistic IO

iO compiles programs into *unintelligible* ones, while *preserving their functionality*.

piO compiles *randomized* programs into *deterministic unintelligible* ones, while *preserving their functionality*.



Why does piO require subexponential iO?

- Programs are only required to be “functionally indistinguishable”

Why does piO require subexponential iO?

- Programs are only required to be "functionally indistinguishable"
- Captures a vast class of programs, e.g.

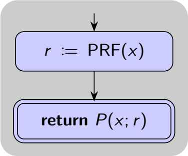
$$\boxed{\begin{array}{l} \underline{P_1(x;r)} \\ \textbf{return } \textsc{Enc}(pk, x; r) \end{array}} \quad \approx \quad \boxed{\begin{array}{l} \underline{P_2(x;r)} \\ \textbf{return } \textsc{Enc}(pk, 0; r) \end{array}}$$

Why does piO require subexponential iO?

- Programs are only required to be "functionally indistinguishable"
- Captures a vast class of programs, e.g.

$P_1(x; r)$		$P_2(x; r)$
return $\textsc{Enc}(pk, x; r)$	$\approx$	**return** $\textsc{Enc}(pk, 0; r)$

- Strategy due to Canetti et al., [CLTV15]:
  - derive random coins from input x via $\mathsf{PRF}(K, x)$

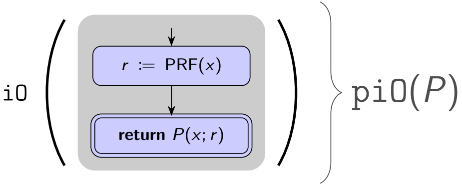


Why does piO require subexponential iO?

- Programs are only required to be "functionally indistinguishable"
- Captures a vast class of programs, e.g.

$P_1(x; r)$		$P_2(x; r)$
return $\textsc{Enc}(pk, x; r)$	$\approx$	**return** $\textsc{Enc}(pk, 0; r)$

- Strategy due to Canetti et al., [CLTV15]:
  - derive random coins from input x via $\mathsf{PRF}(K, x)$



  - use iO to obfuscate this deterministic program

Construction of piO due to Canetti et al., [CLTV15]

piO construction:

$r := \mathrm{PRF}(x)$

return $P(x; r)$

Example:

$P_1(x;r)$		$P_2(x;r)$
return $\textsc{Enc}(pk, x; r)$	$\approx$	**return** $\textsc{Enc}(pk, 0; r)$

- **But** iO security can only be applied if circuits behave fully identically
  - in our example, P_1 and P_2 behave **very differently**

Construction of piO due to Canetti et al., [CLTV15]

piO construction:

$r := \mathrm{PRF}(x)$

return $P(x;r)$

Example:

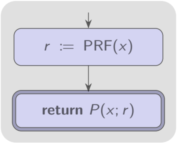
$$\boxed{\begin{array}{l} P_1(x;r) \\ \hline \textbf{return } \textsc{Enc}(pk, x; r) \end{array}} \approx \boxed{\begin{array}{l} P_2(x;r) \\ \hline \textbf{return } \textsc{Enc}(pk, 0; r) \end{array}}$$

- **But** iO security can only be applied if circuits behave fully identically
  - in our example, P_1 and P_2 behave **very differently**

$\rightsquigarrow$ direct (polynomial) reduction to iO won't work

Construction of piO due to Canetti et al., [CLTV15]

piO construction:



Example:

$P_1(x; r)$		$P_2(x; r)$
return $\textsc{Enc}(pk, x; r)$	$\approx$	**return** $\textsc{Enc}(pk, 0; r)$

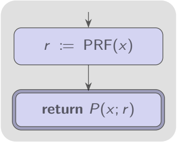
- **But** iO security can only be applied if circuits behave fully identically
  - in our example, P_1 and P_2 behave **very differently**

$\rightsquigarrow$ direct (polynomial) reduction to iO won't work

- Use a "one-input-at-a-time" hybrid argument for all possible inputs
  - this includes the randomness

Construction of piO due to Canetti et al., [CLTV15]

piO construction:



Example:

$P_1(x; r)$		$P_2(x; r)$
return $\textsc{Enc}(pk, x; r)$	$\approx$	**return** $\textsc{Enc}(pk, 0; r)$

- **But** iO security can only be applied if circuits behave fully identically
  - in our example, P_1 and P_2 behave **very differently**

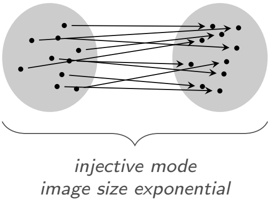
$\rightsquigarrow$ direct (polynomial) reduction to iO won't work

- Use a "one-input-at-a-time" hybrid argument for all possible inputs
  - this includes the randomness

$\rightsquigarrow$ **Our goal:** reduce number of hybrids to a polynomial amount

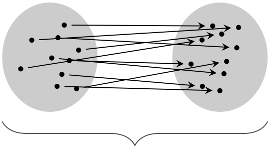
Main tool – Extremely lossy functions

- Extremely lossy functions (ELFs) due to Zhandry, [Zha16] offer two indistinguishable modes:

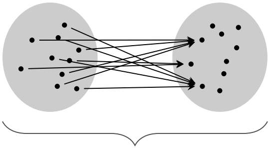


Main tool – Extremely lossy functions

- Extremely lossy functions (ELFs) due to Zhandry, [Zha16] offer two indistinguishable modes:



injective mode
image size exponential

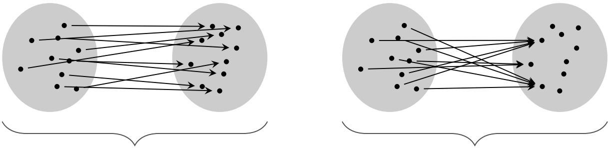


extremely lossy mode
image size **polynomial**

  - exist from exponential DDH

Main tool – Extremely lossy functions

- Extremely lossy functions (ELFs) due to Zhandry, [Zha16] offer two indistinguishable modes:



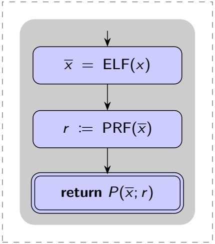
injective mode
image size exponential

extremely lossy mode
image size **polynomial**

  - exist from exponential DDH

- We believe that some sort of (sub)exponential assumption is inherent for probabilistic `iO`
  - ELFs can be used to **push** this subexponentiality to a much more well-understood assumption

First try

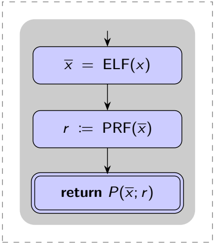


Example:

$P_1(x; r)$		$P_2(x; r)$
return $\textsc{Enc}(pk, x; r)$	$\approx$	**return** $\textsc{Enc}(pk, 0; r)$

- **First try:** reduce number of hybrids by applying the ELF on the input x

First try



Example:

$$\boxed{\begin{array}{l} P_1(x; r) \\ \hline \textbf{return } \textsc{Enc}(pk, x; r) \end{array}} \approx \boxed{\begin{array}{l} P_2(x; r) \\ \hline \textbf{return } \textsc{Enc}(pk, 0; r) \end{array}}$$

- **First try:** reduce number of hybrids by applying the ELF on the input x
- **But** pre-processing the program input x with an ELF will not preserve the expected functionality of the circuit

Our observation

- Common ground for many applications of `piO`:

$$\begin{array}{c} \boxed{\mathcal{D}(m)} \\ \Big\downarrow x \\ \boxed{\texttt{piO}(P)} \end{array}$$

inputs x come from distributions $\mathcal{D}(m)$

Our observation

- Common ground for many applications of `piO`:

$$\begin{tikzcd} \boxed{\mathcal{D}(m)} \arrow[d, "x"'] \\ \boxed{\texttt{piO}(P)} \end{tikzcd}$$

inputs x come from distributions $\mathcal{D}(m)$

  - e.g., $\mathcal{D}(m)$ outputs encryptions of m, or, $\mathcal{D}(\cdot)$ samples public encryption keys.

Our observation

- Common ground for many applications of $\texttt{piO}$:

$$\begin{tikzcd} \boxed{\mathcal{D}(m)} \arrow[d, "x"'] \\ \boxed{\texttt{piO}(P)} \end{tikzcd}$$

inputs x come from distributions $\mathcal{D}(m)$

  - e.g., $\mathcal{D}(m)$ outputs encryptions of m, or, $\mathcal{D}(\cdot)$ samples public encryption keys.

- **Approach:** reduce number of hybrids by applying the ELF *on the random tape of $\mathcal{D}$* to **sparsify** inputs

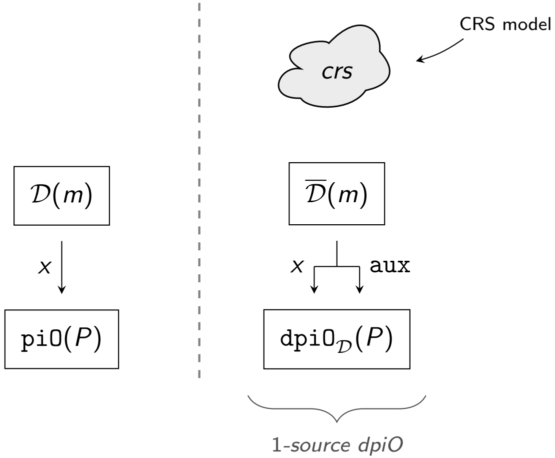
Doubly-Probabilistic IO

- Our framework: **doubly-probabilistic IO (dpiO)**

$$\begin{tikzcd} \boxed{\mathcal{D}(m)} \arrow[d, "x"'] \\ \boxed{\texttt{piO}(P)} \end{tikzcd}$$

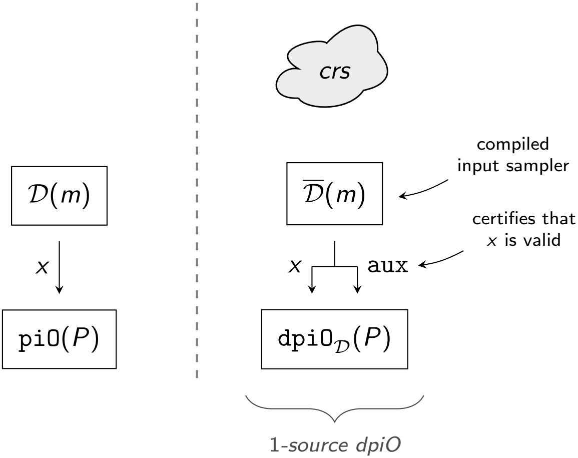
Doubly-Probabilistic IO

- Our framework: **doubly-probabilistic IO (dpiO)**



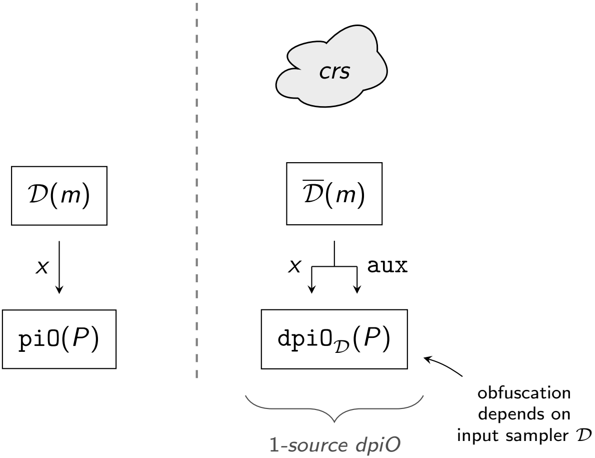
Doubly-Probabilistic IO

- Our framework: **doubly-probabilistic IO (dpiO)**



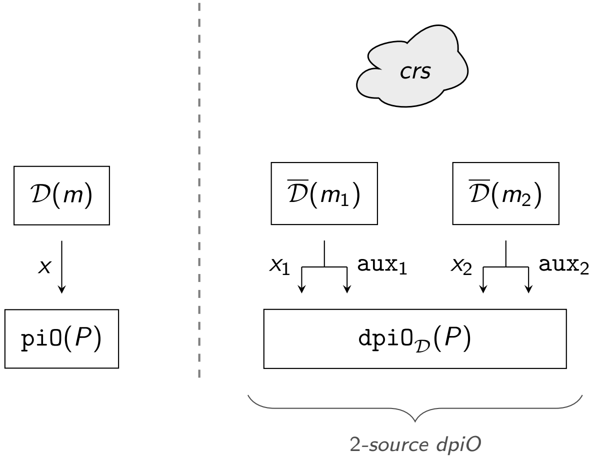
Doubly-Probabilistic IO

- Our framework: **doubly-probabilistic IO (dpiO)**

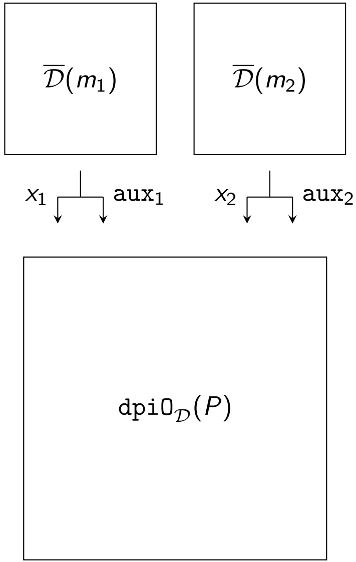


Doubly-Probabilistic IO

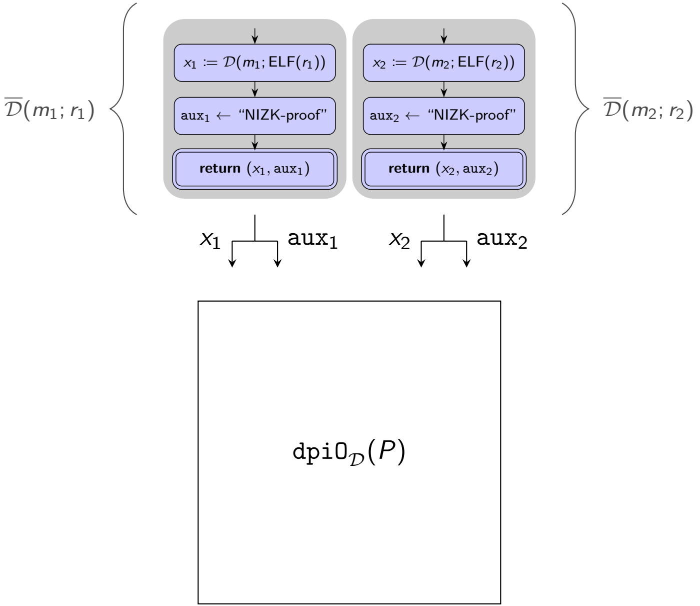
- Our framework: **doubly-probabilistic IO (dpiO)**



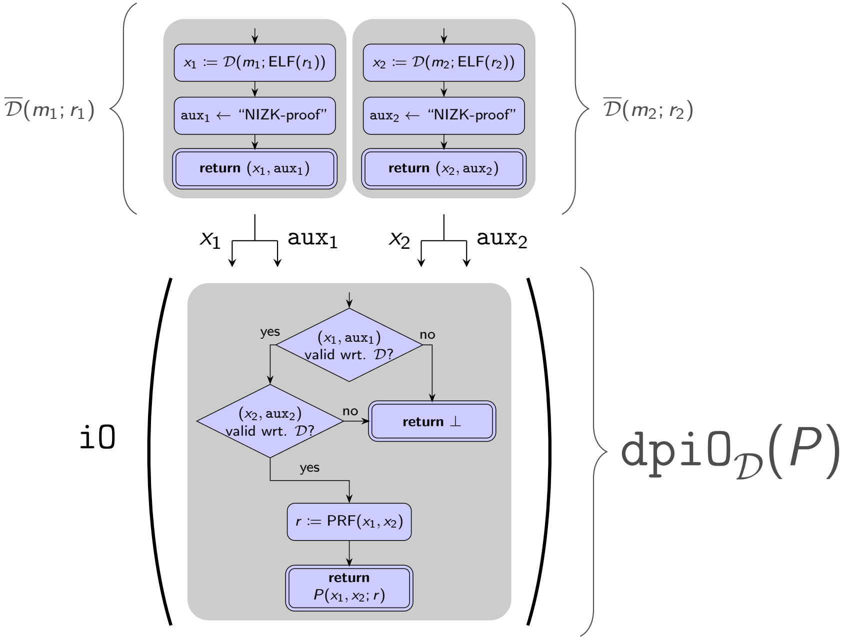
Construction



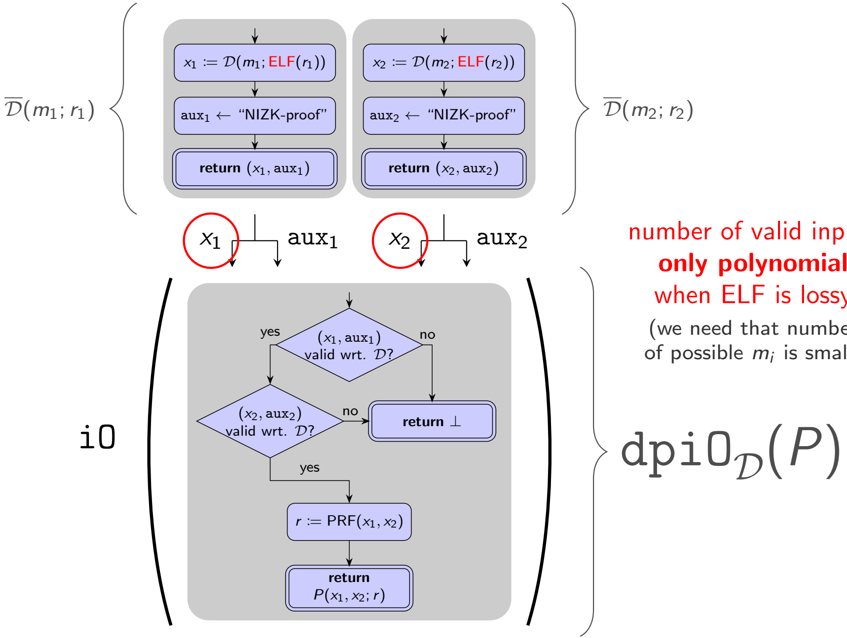
Construction



Construction



Construction



number of valid inputs
only polynomial
when ELF is lossy
(we need that number of possible m_i is small)

Leveled Homomorphic Encryption

- $\mathsf{LHE}_L = (\textsc{Gen}, \textsc{Enc}, \textsc{Dec}, \textsc{Eval})$ is a LHE scheme for depth-L circuits C, if
  - $(\textsc{Gen}, \textsc{Enc}, \textsc{Dec})$ is a PKE scheme, and
  - $\textsc{Eval}$ allows to homomorphically evaluate depth-L circuits on ciphertexts

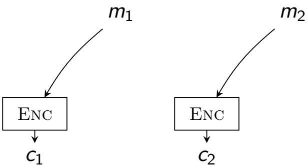
Leveled Homomorphic Encryption

- $\mathsf{LHE}_L = (\textsc{Gen}, \textsc{Enc}, \textsc{Dec}, \textsc{Eval})$ is a LHE scheme for depth-L circuits C, if
  - $(\textsc{Gen}, \textsc{Enc}, \textsc{Dec})$ is a PKE scheme, and
  - $\textsc{Eval}$ allows to homomorphically evaluate depth-L circuits on ciphertexts

m_1 $\qquad\qquad$ m_2

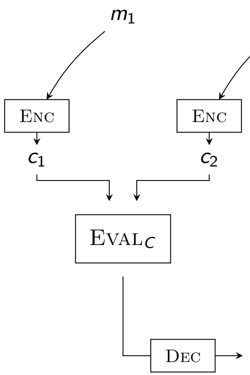
Leveled Homomorphic Encryption

- $\mathsf{LHE}_L = (\textsc{Gen}, \textsc{Enc}, \textsc{Dec}, \textsc{Eval})$ is a LHE scheme for depth-L circuits C, if
  - $(\textsc{Gen}, \textsc{Enc}, \textsc{Dec})$ is a PKE scheme, and
  - $\textsc{Eval}$ allows to homomorphically evaluate depth-L circuits on ciphertexts



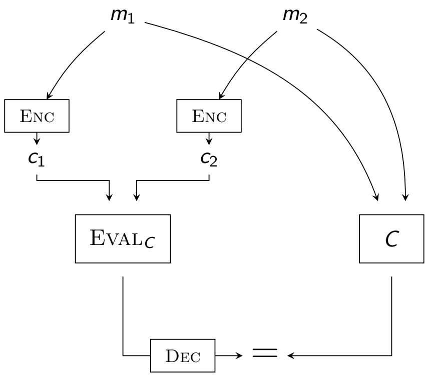
Leveled Homomorphic Encryption

- $\mathsf{LHE}_L = (\textsc{Gen}, \textsc{Enc}, \textsc{Dec}, \textsc{Eval})$ is a LHE scheme for depth-L circuits C, if
  - $(\textsc{Gen}, \textsc{Enc}, \textsc{Dec})$ is a PKE scheme, and
  - $\textsc{Eval}$ allows to homomorphically evaluate depth-L circuits on ciphertexts



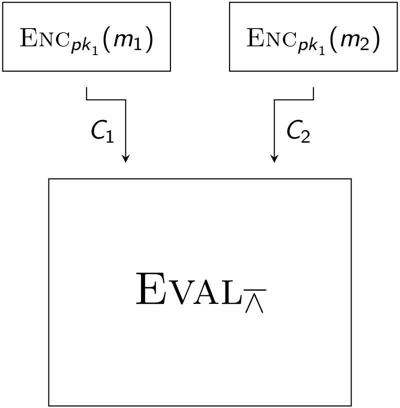
Leveled Homomorphic Encryption

- $\text{LHE}_L = (\textsc{Gen}, \textsc{Enc}, \textsc{Dec}, \textsc{Eval})$ is a LHE scheme for depth-L circuits C, if
  - $(\textsc{Gen}, \textsc{Enc}, \textsc{Dec})$ is a PKE scheme, and
  - $\textsc{Eval}$ allows to homomorphically evaluate depth-L circuits on ciphertexts



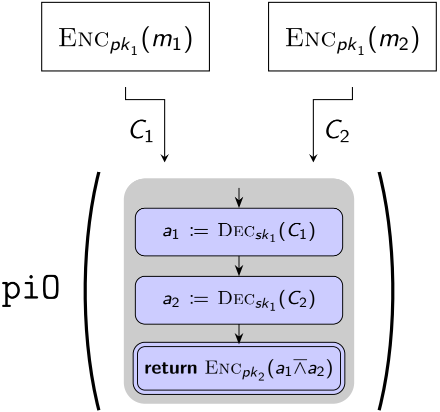
Application LHE/FHE

- LHE construction due to Canetti et al., [CLTV15]
- one NAND gate is evaluated as follows:



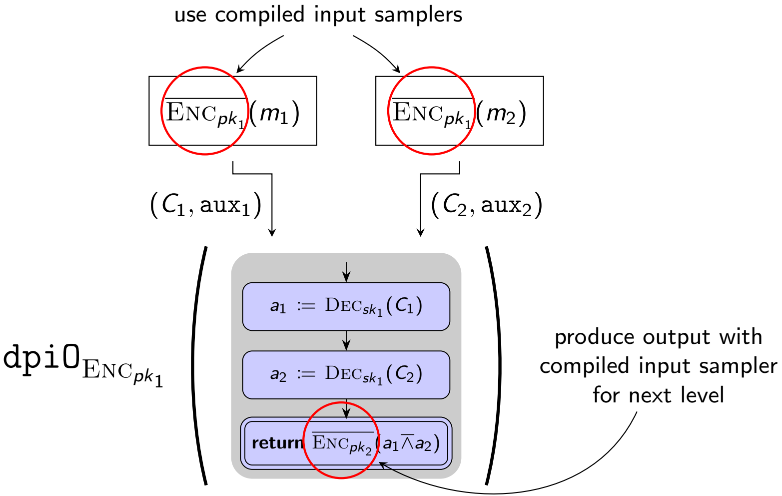
Application LHE/FHE

- LHE construction due to Canetti et al., [CLTV15]
- one NAND gate is evaluated as follows:



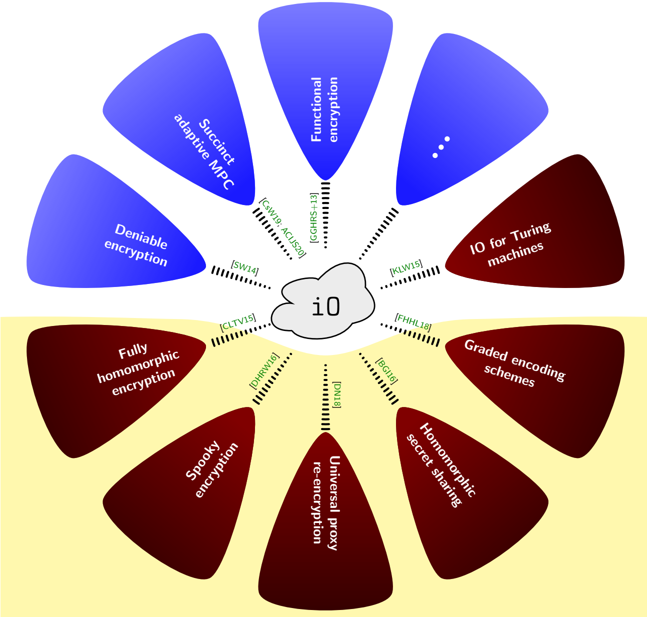
Application LHE/FHE

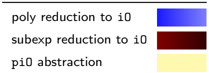
- LHE construction due to Canetti et al., [CLTV15]
- one NAND gate is evaluated as follows:



Conclusion

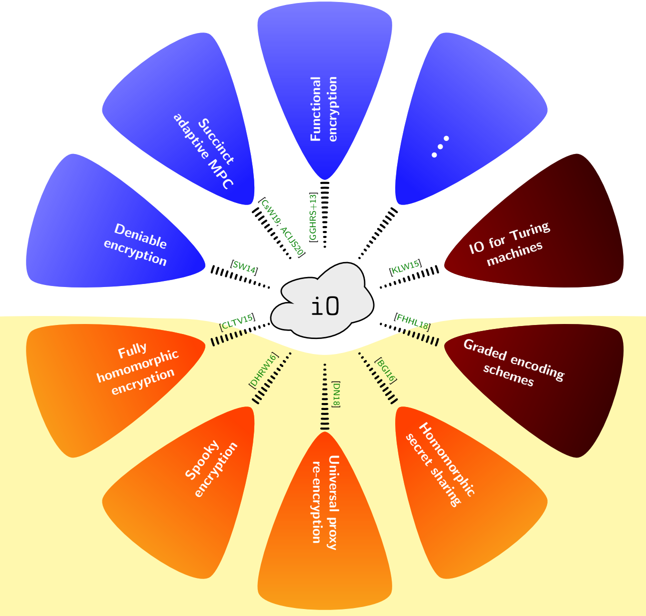
- We can build almost anything from `iO`.
- But what can we do from **polynomial** `iO`?





Conclusion

- We can build almost anything from iO.
- But what can we do from **polynomial** iO?
- And what can we do from **polynomial** iO and **ELFs**?



poly reduction to iO	
subexp reduction to iO	
piO abstraction	
subexp iO plus ELF	